Universidad Carlos III
Escuela politécnica superior

Trabajo de Fin de Grado

Monitorización del rendimiento en una Raspberry Pi

Autor: Álvaro de Lucia Palacios
Tutor: David Expósito Singh

Grado en Ingeniería Informática

LEGANÉS, MADRID
SEPTIEMBRE, 2017

Agradecimientos

Me va a ser imposible no olvidarme de alguna persona que me ha sido de gran ayuda para alcanzar esta meta de terminar mi grado. Han sido cuatro años fantásticos llenos de muchas emociones, a los que pongo un punto y final de una vez por todas. Me alegro de haber conocido a grandes compañeros, a muchos de los cuales les tengo que dar las gracias por haberme ayudado en esta etapa, y otros que han pasado de ser compañeros a ser amigos, como Mario, Guille, Álvaro, Honduco, Juanlu, Alex y Dani (Víctor no le cuento porque le conozco desde que era pequeño y ya sabe que el grado hubiera sido distinto sin esas charlas intercambiando opiniones, aunque ahora seguimos teniéndolas del mundo laboral). No puedo parar de recordar grandes anécdotas y momentos que hemos tenido sin que se me escape una sonrisa, o recordar esos momentos de estrés absoluto en los que piensas que con lo mal que pintaba y al final conseguíamos entregar en esa hora tan famosa en este grado (23:59).

Quisiera también agradecer a todos y cada uno de mis profesores, ya que aunque todos son distintos, todos han aportado su granito de arena para formarme, todos me han dado distintos puntos de vista y distintas maneras de enfocar un problema. En especial tengo que mencionar a mi tutor, David, que ha tenido una paciencia infinita, ya que en mi situación el proyecto iba muy lento, pero siempre ha estado disponible cuando así lo he necesitado.

Dejando aparte la universidad, tengo que agradecer a mis padres que han hecho posible que pueda estudiar el grado. Soy consciente del esfuerzo económico que tuvieron que realizar para que yo pudiera estudiar. Además, siempre han estado apoyándome, aunque siempre que miraban mis apuntes me decían: “Pero, ¿Tu entiendes algo ahí?”. También agradecer a mi hermana, que es un gran apoyo en mi vida personal, lo cual me ayuda a poder centrarme en momentos importantes de mi vida.

Agradecer a mis amigos, que siempre están ahí, y me han ayudado a despejarme en momentos necesarios, lo cual es muy importante en épocas de exámenes sobre todo.

Agradecer a mi chica, Sara, que es la que ha aguantado en primera persona todo el grado. No puedo quejarme de la paciencia que ha tenido, aunque nunca entenderá porque entregaba las prácticas a última hora. Gracias por todo cariño. Aun tengo en mente cuando decidí estudiar el grado, que yo no lo tenía claro, pero mi padre y tú me decíais que lo intentara al menos. Fíjate si no os llego a hacer caso.

Por último, quisiera mencionar a mi hijo, que me ha dado la fuerza necesaria para acabar este proyecto que era lo único que me faltaba. Algún día te recordaré la guerra que me has dado cuando he tenido que acabarlo, que no me dejabas ni 5 minutos seguidos.

Índice de contenidos

Índice de Figuras

Índice de Tablas

Capítulo 1: Introducción

1.1 Motivación

Hoy en día, y cada vez en mayor medida, se está extendiendo el uso de computadoras de placa reducida (SBC), tanto para uso personal, como para uso industrial. Esto se debe a que estas tienen un coste muy reducido en comparación con el resto de computadoras como puedan ser un ordenador de sobremesa, o un portátil.

Las ventajas que presentan estas nuevas tecnologías son que poseen grandes niveles de integración, componentes y conectores reducidos, portable y con un consumo de energía reducido. Esto añadido al coste, hacen que sean una tecnología muy potente, con una relación entre la calidad y el precio muy alta.

Estas computadoras, como pueden ser Raspberry Pi o Arduino, tienen una gran cantidad de usos muy útiles, como puede ser domótica, sistemas embebidos que realizan la función de controlador e interfaz de otro sistema o uso para fines educativos.

Aparte de estas aplicaciones, que son más comunes en el uso de esta tecnología, se han utilizado para otros proyectos mucho más sorprendentes, demostrando que es una tecnología potente y que aún está por explotar. De hecho, el volumen de ventas a superado con creces lo esperado, estando actualmente en una cifra superior a 12.5 millones de unidades vendidas. Esto sitúa a la Raspberry Pi como la tercera plataforma de informática más popular del mundo, por detrás de PC y del Mac. A continuación se muestran algunos proyectos donde se ha utilizado esta plataforma:

Escáner 3D

Este proyecto utiliza 40 Raspberry Pi con 40 Pi Cameras, además de 40 tarjetas SD y 1 fuente de alimentación de 60A y 5V para dar corriente a todas las Raspberry Pi. El tamaño del proyecto es especialmente destacable, ya que es capaz de escanear a cualquier persona, obteniendo unos resultados muy precisos.

Figura 1 - Proyecto escáner

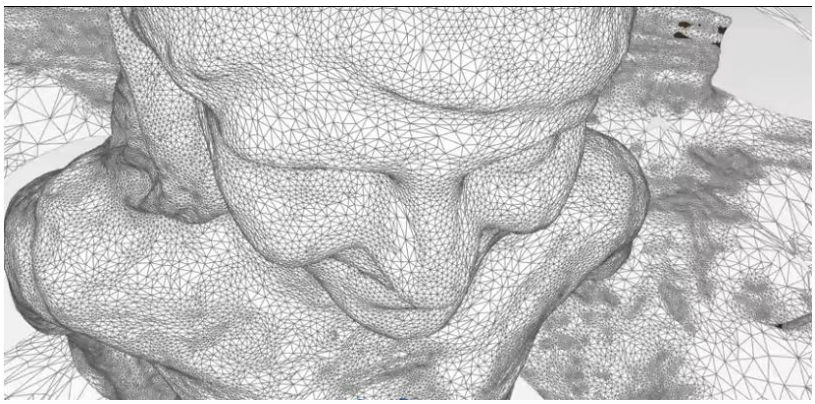


Figura 2 - Resultado proyecto escáner

Estación meteorológica

Este proyecto pretende crear una estación meteorológica con una Raspberry Pi, como se muestra en la imagen. El sistema es capaz de mostrar todo tipo de información: temperatura, humedad, presión del aire, niveles de luz y radiación ultravioleta, niveles de monóxido de carbono o de dióxido de nitrógeno, etc., y todo ello para luego ser compartido con nuestros dispositivos vía Internet.

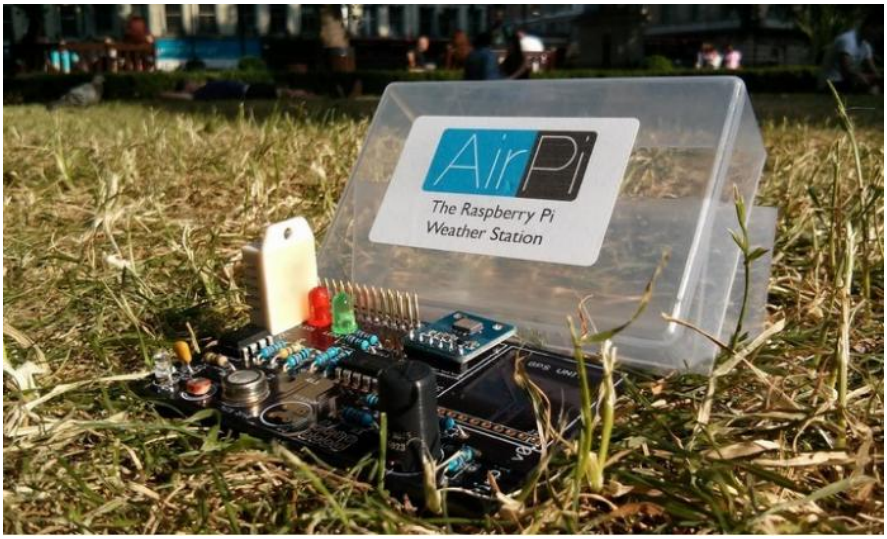


Figura 3 - Proyecto estación meteorológica

Por último, esta tecnología cuenta con mucha información en Internet, así como distintos sistemas operativos y periféricos, lo cual nos ayuda bastante a la hora de afrontar los problemas de trabajar con una tecnología relativamente nueva.

1.2 Objetivos

1.2.1 Objetivo Principal

El objetivo principal del proyecto es realizar un planificador, que sea capaz de organizar la secuencia de ejecución de los procesos que se comunican con un servidor, mejorando el rendimiento del sistema cuando utiliza el planificador por defecto.

Hacer un estudio del rendimiento de la plataforma, así como un planificador para mejorar su rendimiento, puede ser un proyecto muy interesante, dado que puede ser útil para futuras aplicaciones en las que se utilice la consola como un servidor, o un sistema que reciba peticiones, y necesite una ejecución eficiente, ya que este es un punto crítico en todas las aplicaciones cliente-servidor.

Esta mejora de rendimiento se traduce en reducción del tiempo total empleado para ejecutar todos los procesos.

Por último, la granularidad del planificador que se va a desarrollar será de grano grueso. Esto se debe a que un planificador de grano fino se ejecuta en rodajas de tiempo muy pequeñas (normalmente hablamos de milisegundos) siguiendo la planificación "round robin". El trabajo se dividiría en pequeños procesos que serían repartidos entre los procesadores del sistema, los cuales ejecutarían estas pequeñas tareas. Esto por lo tanto requeriría un mayor número de comunicaciones y sincronización, lo cual genera una mayor sobrecarga. Por este motivo, el planificador será de grano grueso, utilizando intervalos largos para la ejecución de los procesos, como se podrá observar en los tiempos de ejecución de cada proceso en las pruebas. De esta manera reducimos la sobrecarga generada en el sistema.

1.2.2 Objetivos Específicos

- Realizar un análisis de los contadores hardware existentes en la plataforma para conocer cuáles son aquellos contadores que influyen en mayor o menor medida cuando se consumen muchos recursos de memoria, CPU o entrada/salida del sistema. De esta forma, podremos saber cuáles nos serán útiles para categorizar los procesos entrantes en función de los valores obtenidos en los contadores.

- Analizar el sistema y sus especificaciones hardware, para determinar el número de procesos en ejecución más adecuado, maximizando el rendimiento del sistema, sin llegar a penalizarlo por la sobrecarga de tener demasiados procesos en ejecución. También se evaluará que tipos de procesos conviven mejor en ejecución, para establecer un orden de preferencia entre los procesos.

- Establecer en el planificador un orden por prioridades, ejecutando antes los procesos que tengan un valor de prioridad mayor. Para los procesos que compartan el mismo nivel de prioridad, se escogerá aquel que lleve más tiempo esperando, o lo que es lo mismo, un planificador 'FIFO' con prioridad.

- Ser capaces de modularizar la aplicación, separándola en dos módulos claramente diferenciados, como son el primer módulo que se compone de los procesos y los clientes, y un segundo módulo compuesto por la relación entre el cliente y el servidor del sistema.

1.3 Estructura del documento

Este documento está estructurado en distintos capítulos en los cuáles se tratan diferentes aspectos relativos al proyecto. En concreto consta de un total de 10 capítulos más 4 anexos. El contenido de los capítulos se explica a continuación.

- Capítulo 1: Introducción: Contiene la motivación, los objetivos del proyecto y la estructura del documento.

- Capítulo 2: Estado del arte: Estudia las diferentes herramientas relacionadas con la que se ha usado en este proyecto, los contadores hardware de la plataforma usada y los planificadores más comunes.

- Capítulo 3: Diseño: Contiene las especificaciones del hardware de la plataforma y la arquitectura propuesta para nuestro sistema.

- Capítulo 4: Implementación: Incluye los algoritmos y las lógicas implementadas en nuestro sistema, además del protocolo y las señales que se han utilizado.

- Capítulo 5: Análisis: Identifica los requisitos de nuestro sistema.

- Capítulo 6: Evaluación y pruebas: Incluye la batería de pruebas utilizada para comprobar la aplicación y la validación de modelo desarrollado.

- Capítulo 7: Contexto de desarrollo: Incluye el marco regulador y el entorno socio-económico del proyecto.

- Capítulo 8: Planificación: Contiene la planificación temporal del proyecto.

- Capítulo 9: Presupuesto: Incluye el presupuesto del proyecto.

- Capítulo 10: Conclusiones y trabajos futuros

Por último, los anexos incluyen los manuales de instalación y usuario, además de un resumen en inglés del proyecto realizado. La bibliografía se compone de los documentos o URLs de donde se ha sacado información.

Capítulo 2: Estado del arte

2.1 Herramientas de monitorización de aplicaciones

Actualmente, existen numerosas herramientas para monitorizar el rendimiento de un sistema. Todas estas herramientas, además de la función de monitorización del sistema, suelen tener funcionalidades adicionales, como pueden ser, por ejemplo, la función de depurado o la función de análisis de trazas de ejecución de programas para detectar “cuellos de botella".

A continuación vamos a mostrar algunas de estas herramientas:

2.1.1 SAR

Esta herramienta te permite monitorizar en tiempo real del rendimiento del sistema **[21]**. También permite recoger los datos del rendimiento del sistema, de manera continua, para poder localizar de esta manera posibles “cuellos de botella”. Entre todas sus funciones, nos permite:

- Monitorizar el uso de CPU.
- Analizar las estadísticas de CPU.
- Analizar memoria en uso y disponible.
- Intercambiar espacio usado y disponible.
- Recoger actividad global de E/S del sistema.
- Obtener actividad individual de E/S de un dispositivo.
- Obtener estadísticas de los cambios de contexto.
- Recoger estadísticas de la red.

2.1.2 Tcpdump

Esta herramienta nos permite analizar los paquetes de la red **[22]**. Con esta funcionalidad, podríamos capturar los paquetes que llegan a un determinado puerto o capturar los paquetes que se intercambian en una comunicación TCP. De esta manera, podemos localizar posibles “cuellos de botella” que penalizan el rendimiento de una aplicación o sistema.

2.1.3 Nagios

Esta es una herramienta de código libre, que permite monitorizar infraestructuras **[23]**. De esta manera, podemos controlar cuando se *cae* un servidor, o una base de datos, para mandar un mensaje al administrador de la misma. También podría notificar cuando un disco supera un porcentaje de su capacidad total. Las siguientes son algunas de las infraestructuras que es capaz de monitorizar:

- Cualquier hardware (servidor, *router*, *switch*…).
- Servidores Windows y Linux.
- Bases de datos.
- Servicios del sistema (email, nfs…).
- Servidores web.
- Aplicaciones propias de usuario.

2.1.4 Iostat

Esta herramienta nos permite obtener la información de la CPU o de la E/S a disco **[24]**. Permite monitorizar todas las particiones del sistema o un dispositivo específico.

2.1.5 Mpstat

Esta herramienta nos permite obtener toda la información relativa al procesador **[25]**.

2.1.6 Vmstat

Esta herramienta nos permite obtener toda la información relativa a la memoria virtual del sistema **[26]**.

2.1.7 PS

Esta herramienta nos muestra los procesos que están corriendo en el sistema **[27]**, junto con información adicional que nos puede servir para hacer una evaluación individual de cada proceso. Además, nos permite filtrar los procesos por *username* o por jerarquía, entre otros.

2.1.8 Free

Esta herramienta nos muestra la información de la memoria física (RAM) y la memoria virtual **[28]**. Nos muestra tanto la memoria en uso, como la memoria disponible.

2.1.9 Strace

Esta es una herramienta de depurado de aplicaciones **[29]**. Nos muestra las llamadas al sistema de un proceso/programa, así como las señales recibidas. Es una herramienta muy potente, sobre todo cuando no se dispone del código fuente.

2.1.10 Lsof

Esta herramienta nos permite monitorizar todos los archivos abiertos en el sistema **[30]**. Entre estos, se incluye las conexiones de red, los dispositivos y los directorios del sistema. Como en muchas de las herramientas, también permite realizar un filtrado para acotar el resultado obtenido.

Como hemos podido observar, hay numerosas herramientas para la monitorización de aplicaciones. Son herramientas muy variadas, que nos permiten tener un control de nuestro sistema.

2.2 Contadores hardware

Nuestra aplicación hará uso de los contadores hardware de la arquitectura, para monitorizar los procesos de la aplicación. Para esta captura de eventos hardware se han utilizado las siguientes herramientas:

2.2.1 PAPI:

Esta librería nos sirve como herramienta de monitorización **[2]**. Nos proporciona una serie de componentes para medir el rendimiento de una aplicación. Es capaz de realizar mediciones tanto de la parte hardware como de la parte software de la aplicación.

PAPI se ha ido actualizando para ser compatible con las nuevas plataformas, y de esta manera, han intentado que todas las funciones estuvieran disponibles para el mayor número de plataformas.

Sin embargo, una vez se instaló la librería en la Raspberry Pi, ejecutamos una serie de pruebas para cerciorarnos de que era capaz de monitorizar los elementos necesarios para seguir adelante con el desarrollo de nuestra aplicación. Desgraciadamente, la versión actual de la herramienta no era capaz de monitorizar los contadores hardware de la plataforma, cuyas características se especifican en la sección 3.1 del presente documento, de manera que las funciones que eran compatibles no nos sirvieron y tuvimos que descartar la opción de utilizar esta herramienta.

2.2.2 Perf:

Esta es una de las herramientas de monitorización para Linux **[3]**. Esta herramienta es capaz de monitorizar contadores hardware y *tracepoints,* entre otros elementos. Está incluida en el kernel de Linux y frecuentemente se actualiza y se mejora.

Ha sido la opción escogida debido a que es compatible con la plataforma con la que hemos trabajado. Además, es capaz de monitorizar los contadores de la caché, de las tablas de paginación TLB y otros eventos que nos eran de gran utilidad. Entre los contadores y eventos que es capaz de monitorizar están los siguientes, algunos de los cuáles han sido utilizados para el desarrollo de este proyecto:

- L1-dcache-loads: Número de accesos de lectura a caché nivel 1.
- L1-dcache-load-misses: Número de fallos de acceso de lectura a caché nivel 1.
- L1-dcache-stores: Número de accesos de escritura a caché nivel 1.
- L1-dcache-store-misses: Número de fallos de acceso de escritura a caché nivel 1.
- dTLB-load-misses: Número de fallos de acceso de lectura a caché TLB.

- dTLB-store-misses: Número de fallos de acceso de escritura a caché TLB.
- BRANCH-INSTRUCTIONS: Número de instrucciones de salto.
- BRANCH-MISSES: Número de fallos de instrucciones de salto.
- CPU-CYCLES: Número de ciclos de CPU.
- INSTRUCTIONS: Número de instrucciones de CPU.
- CONTEXT-SWITCHES: Número de cambios de contexto.
- PAGE-FAULTS: Número de fallos de página.
- CPU-CLOCK: Frecuencia de reloj del procesador.

Además, esta herramienta no solo nos permite monitorizar un evento en concreto, si no que podemos indicarle el tiempo que queremos monitorizarlo, así como indicarle el evento por PID. Tiene muchas más opciones de filtrado, pero estas han sido las más útiles para el desarrollo de nuestra aplicación. Por este motivo, se realizaron una serie de pruebas, y tras comprobar que el funcionamiento era el esperado, escogimos esta opción. A continuación vamos a mostrar algunos de sus comandos más utilizados:

- **Perf record**: registra eventos para posteriormente informar acerca de los datos obtenidos.

- **Perf report**: realiza un desglose de los eventos por proceso, función, etc.

- **Perf top**: muestra los contadores de los eventos en el momento.

- **Perf list**: Nos muestra todos los elementos que es capaz de monitorizar esta herramienta. En función de la plataforma donde se ejecute, cambian los elementos que es capaz de monitorizar.

- **Perf bench**: ejecuta *microbenchmarks* del kernel.

- **Perf stat**: Este ha sido uno de los comandos más utilizados. Nos permite monitorizar los contadores hardware de un determinado evento.

2.3 Planificadores de procesos

Esta es una de las partes más importante de nuestra aplicación, debido a que el planificador del servidor será el encargado de continuar/parar los procesos entrantes, de manera, que la sobrecarga del sistema, así como el acople con otros procesos dependerá del tipo de planificación escogida.

Para la elección del tipo de planificador, así como para su desarrollo, tuvimos en cuenta los planificadores más utilizados, para poder realizar una valoración sobre sus ventajas/desventajas.

Los planificadores más comunes son los siguientes:

- FCFS (*First come first served*): El proceso que se ejecuta es el primero que llega, y el resto se van encolando siguiendo este patrón.

- SJF (*Shortest Job First*): Los procesos tienen un tiempo asignado, y se ejecutará aquel que tenga un tiempo menor. En caso de que dos procesos tengan el mismo tiempo, se empleará el algoritmo FCFS.

- Round Robin: Cada proceso tiene asignado un tiempo de ejecución, que es el mismo que tiene asignado el resto de procesos.

- Planificacion por prioridad: El proceso que se ejecuta es aquel que tiene mayor prioridad.

Debido a los distintos tipos de planificadores y a las necesidades existentes en nuestro sistema de ejecución de procesos, se realizó un análisis del sistema, para decidir que planificador podría ser más beneficioso. Además, nos apoyamos en dos documentos, el primero **[6]** es un 'Technical report' publicado en la universidad de Minnesota en junio de 2003 y el segundo **[7]** es un artículo publicado en Las Vegas en diciembre de 2015, de donde hemos podido obtener otro tipo de ideas, para acabar definiendo nuestro planificador de la aplicación.

La decisión tomada a partir de la información recopilada fue que, dado que tenemos distintos tipos de procesos, los cuales se acoplan mejor o peor con los otros tipos, nuestro planificador debería tener en cuenta el tipo de proceso que hay en ejecución. Además, dado que nuestra plataforma solo cuenta con un núcleo, había que maximizar el rendimiento añadiendo procesos en ejecución, hasta el punto de que el balance entre el número de procesos y la sobrecarga generada no penalizara a nuestra aplicación. Además, se decidió añadir prioridad a los procesos, de manera que, se ejecutarían los procesos más prioritarios, predominando el tipo de proceso primero.

Por último, el planificador se sometió a pruebas, para comprobar que efectivamente, era un planificador que mejoraba el rendimiento de la aplicación. Estas pruebas nos podrían ser de gran ayuda para determinar el número de procesos más adecuado en ejecución.

Capítulo 3: Diseño

3.1 Descripción de plataforma: hardware

La plataforma donde se ha desarrollado el proyecto, y por tanto, la aplicación, ha sido una Raspberry Pi modelo B con las siguientes características principales:

- Chip: Broadcom BCM 2835
- CPU: ARM1176JZF-S a 700MHz
- GPU: VideoCore IV a 250 MHz
- RAM: 512MB
- Entrada de video: Cámara CSI
- Salida de video: HDMI 1.4
- Puertos USB: 2
- Tarjeta de almacenamiento: SD

Estas características han ido mejorando conforme han ido saliendo a la venta nuevos modelos de Raspberry Pi. Por ejemplo, las últimas versiones de Raspberry Pi cuentan con procesadores de hasta 4 núcleos. En nuestro caso, nuestro procesador cuenta con un único núcleo y dos hilos de CPU.

Gracias a estas limitaciones hardware surgió la idea de que un planificador de procesos podría suponer un gran avance respecto al rendimiento en la plataforma, ya que la sobrecarga de la misma genera un rendimiento muy malo de la misma.

3.2 Arquitectura del sistema

3.2.1 Entorno de desarrollo

En el entorno de desarrollo de nuestra aplicación hemos utilizado un lenguaje de programación C, ya que consideramos que era un lenguaje adecuado porque se utiliza en programación de sistemas y se había estudiado en el grado y había una base de conocimiento importante que nos podían ayudar a agilizar el proceso de desarrollo.

El sistema operativo con el que hemos trabajado ha sido Raspbian. Este es un sistema operativo basado en Debian, que está precompilado y tiene una instalación muy sencilla en la plataforma. Además, este sistema operativo ofrece un rendimiento excelente en este tipo de plataforma, lo que nos hizo declinarnos por esta opción.

Por último, la herramienta de monitorización utilizada ha sido ‘Perf’, la cual hemos explicado en la sección 2.2.2 y no queremos incidir de nuevo en ella. Solamente ha hecho falta utilizar esta

herramienta, ya que queríamos monitorizar los procesos, y no necesitábamos usar otras herramientas que fueran más complejas y que nos ofrecían los mismos resultados. Por otra parte, para programar en C, hemos hecho uso de ficheros de texto plano, no se ha utilizado ningún entorno adaptado para ello.

3.2.2 Familias de aplicaciones

Con el objetivo de analizar el rendimiento de nuestra plataforma, se necesitaba someterla al máximo estrés posible. Para ello, se utilizarán benchmarks de distintos tipos. En nuestro caso, vamos a utilizar 3 tipos de benchmarks:

- CPU
- Memoria
- Entrada/salida

Cada uno de estos tipos ejecutará una serie de cargas de trabajo, orientadas a aumentar el estrés de una parte de la aplicación o de otra. En concreto, la carga de trabajo realizada por cada tipo es la siguiente:

- CPU: Realiza un número elevado de cálculos aritméticos en base a unos números aleatorios que genera para aumentar el uso de CPU.

- Memoria: Escribe o lee bloques de memoria de un tamaño determinado, hasta un máximo establecido. Estos bloques se pueden variar en función de las opciones escogidas al ejecutar el benchmark.

- I/O: Abre un fichero para leer o escribir bytes o bloques de bytes. La cantidad de ficheros para leer/escribir se pueden predefinir utilizando las opciones existentes al ejecutar el benchmark, así como también se puede escoger el tamaño del bloque de bytes.

Estas aplicaciones de los distintos tipos de benchmarks son de código libre, están programadas en C y han sido obtenidas del repositorio de github con referencia **[5]** en esta sección del documento. Gracias a estos benchmarks hemos sido capaces de medir el rendimiento de nuestra plataforma de distintas maneras. Estas maneras han sido las siguientes:

- Combinando los distintos tipos para que se ejecuten a la vez y analizar su acoplamiento con el resto de benchmarks en ejecución.

- Ejecutando los benchmarks de manera serial o todos al mismo tiempo. También se ejecutarán por grupos una vez analizado su nivel de acoplamiento.

- Aumentando el número de benchmarks ejecutados a la vez, para llevar la plataforma al límite de su rendimiento, para ver como respondía en casos extremos.

3.2.3 Arquitectura propuesta

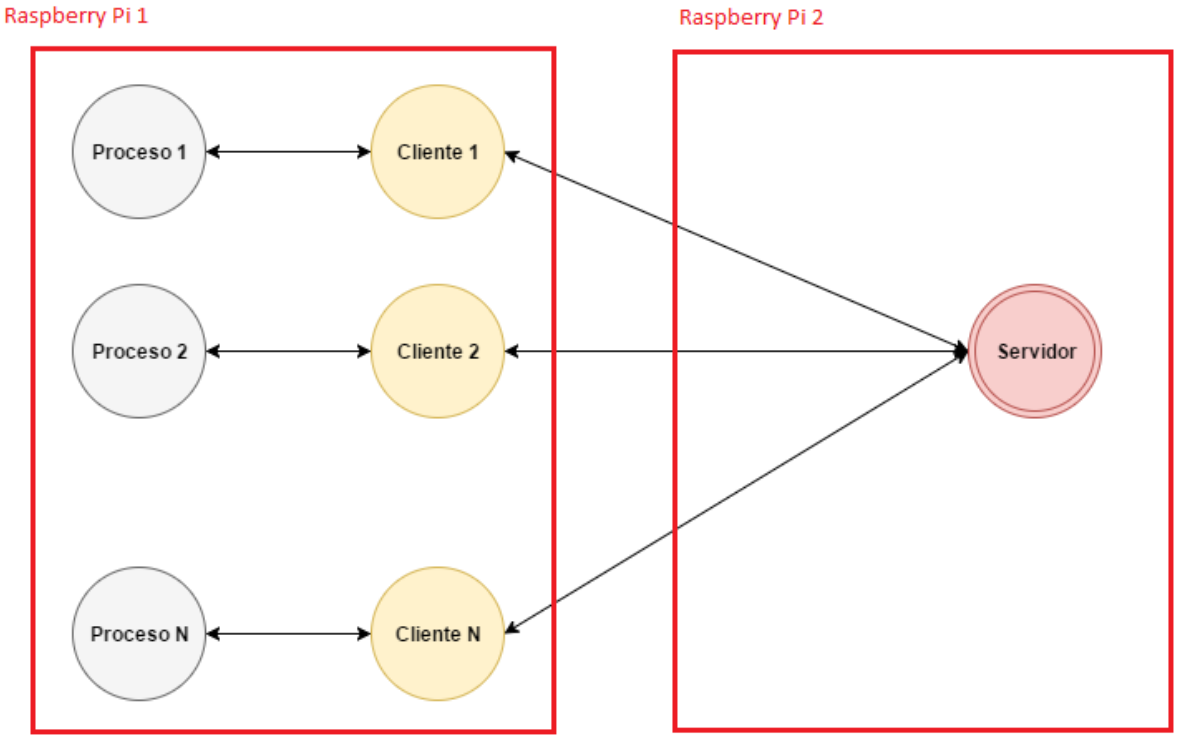


Figura 4 - Arquitectura aplicación

La figura 4 muestra la arquitectura de nuestra aplicación. Como se puede observar la aplicación se podría separar en dos módulos claramente diferenciados, los cuales correrían en diferentes Raspberry Pi.

El primer modulo consta de los diferentes procesos de nuestra aplicación, junto con los clientes de la misma.

El segundo modulo estaría compuesto por la relación entre los clientes de la aplicación y el servidor de la misma.

3.2.3.1 Aplicación en ejecución

En la aplicación habrá un número genérico de procesos, sin ninguna restricción en el límite (exceptuando las restricciones de rendimiento propias de la plataforma, que eso dependerá de la máquina donde se corra la aplicación).

Por otra parte, habrá el mismo número de clientes que de procesos, ya que cada cliente se encargará individualmente de monitorizar cada proceso. Por lo tanto, habrá N procesos y N clientes.

Cada proceso será el encargado de ejecutar un *benchmark* determinado (siempre comprendido entre los tipos de *benchmarks* seleccionados), de manera, que consumirá más recursos de un tipo.

3.2.3.2 Biblioteca de monitorización

La monitorización de los procesos será realizada por parte de los clientes. Los clientes, en función del PID, monitorizarán un único proceso durante un tiempo determinado.

Durante esta monitorización, se utilizará la herramienta *perf*, para obtener el valor de los distintos contadores hardware del sistema, así como otros datos importantes para tomar las decisiones necesarias con el objetivo de mejorar el planificador.

Una vez el cliente ha realizado la monitorización del proceso, se encargará de conectarse con el servidor, y mandarle aquellos datos requeridos por este. Entre estos datos, también le enviará el PID del proceso que ha monitorizado, de manera que el cliente puede terminar su ejecución, y el servidor sería el encargado directo de manejar el proceso como considere oportuno.

Esta decisión de diseño de que el cliente termine su ejecución y el servidor se encargue directamente del manejo del proceso, se debe a que de esta manera reducimos la sobrecarga del sistema, ya que habría un proceso menos en ejecución, y además, ahorramos un paso a la hora de detener/arrancar/eliminar el proceso si el servidor así lo considerase oportuno.

El comando que hemos utilizado para monitorizar desde el cliente ha sido el siguiente:

Perf stat –p PID **–e** Eventos Capturados **–a** Tiempo Monitorización

Los procesos por su parte obtienen su PID y lo vuelcan a un fichero, donde se añadirán todos los PIDs de todos los procesos existentes. Ese fichero será de donde los clientes saquen el identificador del proceso que van a monitorizar. Una vez hecho esto, los procesos realizan una llamada al sistema, ejecutando un tipo de benchmark determinado.

En la figura 5 mostraremos una imagen donde se puede ver el flujo que se produce en la conexión entre el proceso y el cliente:

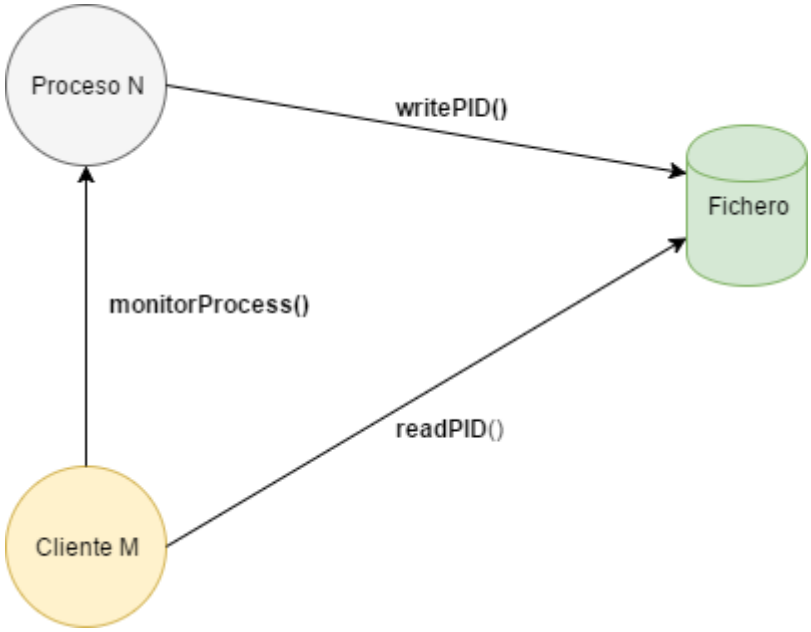


Figura 5 - Flujo proceso/cliente

El flujo consiste en:

1. El proceso escribe su PID en un fichero.
2. El cliente lee el PID del fichero.
3. El cliente monitoriza el proceso

3.2.3.3 Módulo de comunicación

La comunicación entre el cliente y el servidor se realiza a través de sockets. El cliente le proporciona al servidor aquellos datos que le son necesarios para continuar con la monitorización de los procesos, así como para mejorar el rendimiento del planificador.

El servidor está escuchando constantemente, de manera que cuando un cliente se comunica con él, creará un hilo para tratar la petición de ese cliente, y así poder seguir esperando otra comunicación y de esta manera no bloquear el servidor mientras se está comunicando con un cliente. Además, de esta manera se consigue que si en algún momento un cliente le manda un dato erróneo o se queda bloqueado y no es capaz de mandarle datos, el servidor no dejará de funcionar, si no que es el hilo el que fallará y no alterará el funcionamiento de la aplicación.

Por otra parte, el servidor creará otro hilo, que será el encargado de realizar las funciones del planificador, deteniendo aquellos procesos que considere oportuno, y arrancándolos cuando considere necesario. Este hilo decidirá que procesos se ejecutan en función de la configuración establecida.

Por último, el hilo encargado de tratar las peticiones de cliente será el encargado de recibir los datos del cliente, y en función de estos, categorizar de qué tipo es el proceso monitorizado por el cliente, y añadir el PID del proceso a su vector correspondiente, donde se encontraran todos los PID de los procesos de ese tipo monitorizados por el servidor.

Para evitar los problemas de concurrencia, ya que la aplicación puede configurarse *multihilo*, para el acceso a los vectores de procesos se utilizará un *mutex*. Este *mutex* bloqueará el acceso de escritura al vector, para evitar inconsistencias si dos hilos accedieran al vector al mismo tiempo y escribieran su PID en la misma posición de memoria. De esta manera, evitaremos los problemas de condiciones de carrera.

En la figura 6 se muestra el flujo que se produce en la conexión entre los distintos clientes y el servidor:

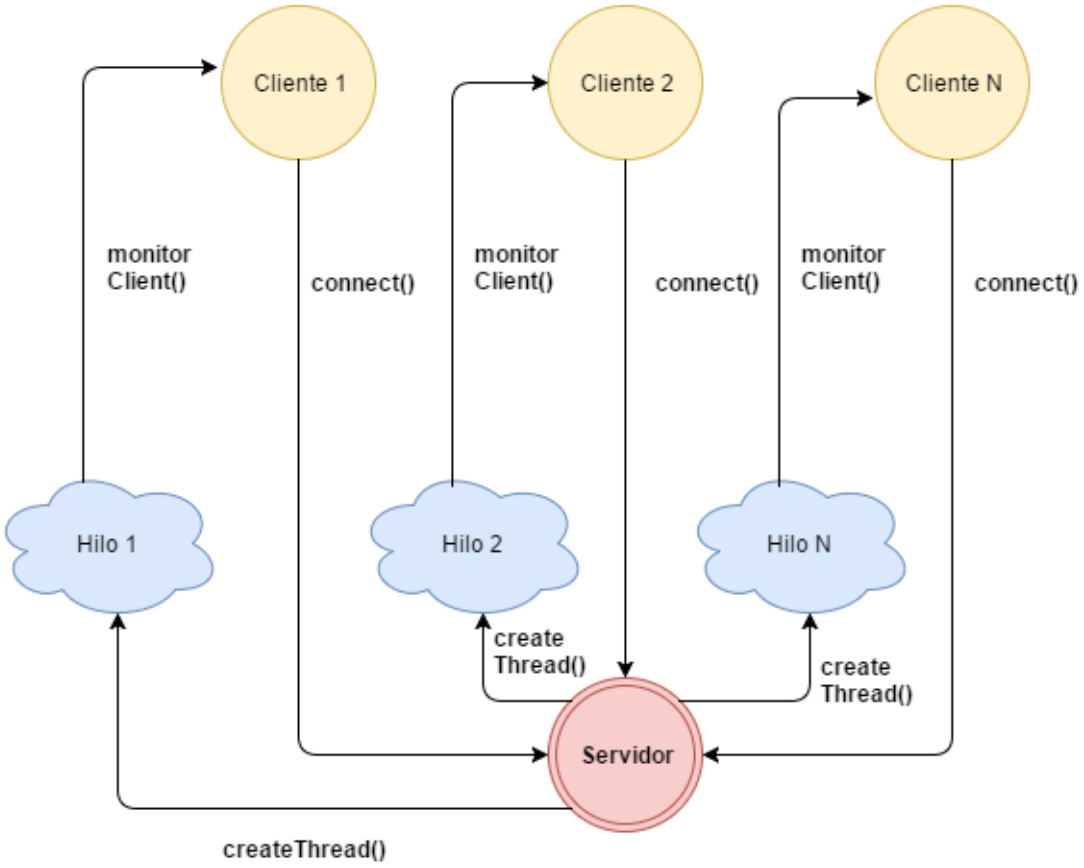


Figura 6 - Conexión cliente/servidor

El flujo sería el siguiente:

1. El cliente establece la conexión con el servidor.
2. El servidor crea un hilo independiente.
3. El hilo se encarga de tratar la petición de cliente.

A continuación, en la figura 7 se muestra el flujo existente en la conexión entre el cliente y el hilo creado por el servidor:

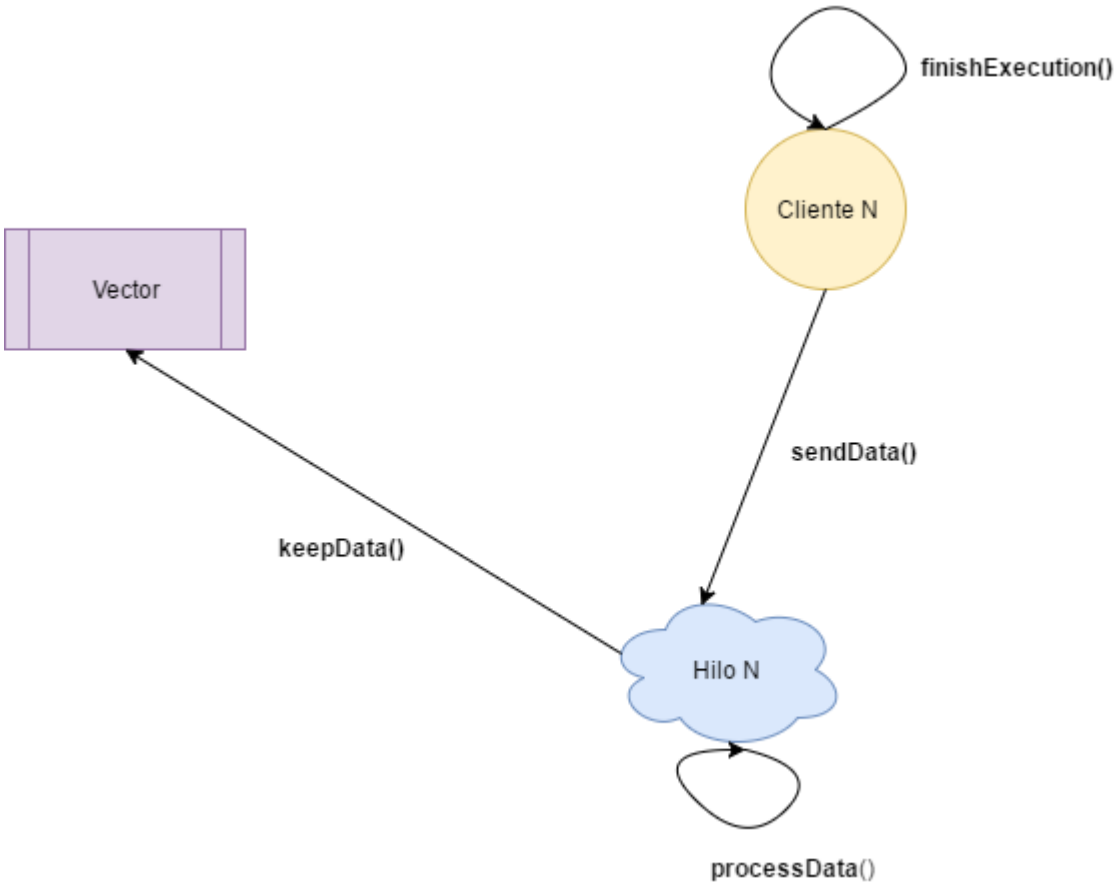


Figura 7 - Conexión cliente/hilo petición

El flujo sería el siguiente:

1. El cliente le envía los datos al hilo encargado de tratar la petición. Entre estos datos estará el PID, el valor de los contadores hardware y la prioridad del proceso.

2. El hilo procesa los datos recibidos, para establecer qué tipo de proceso.

3. El hilo añade los datos al vector, teniendo en cuenta la prioridad del proceso, ya que se introducen los datos en el vector siguiendo un orden.

4. El cliente termina su ejecución.

Con esto terminaríamos la comunicación cliente-servidor. Ahora sería el servidor el encargado de planificar los procesos existentes, y el cliente terminaría su ejecución, reduciendo la sobrecarga de la plataforma, ya que si fuera el cliente el encargado de monitorizar el proceso, tendría que haber más comunicaciones entre el cliente y el servidor, además de un mayor número de hilos ya que estos hilos que corresponden a los clientes seguirían ejecutándose, generando así una mayor sobrecarga del sistema.

Capítulo 4: Implementación

4.1 Módulo de caracterización

Una parte muy importante a la hora de realizar el planificador era escoger que eventos nos eran más útiles en el momento en que se intentara diferenciar los tipos de procesos. Para ello, se ha intentado realizar una diferenciación clara, sin tener que realizar un envío exhaustivo de todos los eventos disponibles que se indicaron en la sección 2.2.2 de este documento.

Por este motivo, se realizaron empíricamente pruebas exhaustivas con todos los eventos existentes y se concluyó que los eventos que se muestran a continuación son los más relevantes para realizar una correcta categorización de los distintos tipos de benchmarks:

- L1-dcache-loads: Número de accesos de lectura a memoria caché nivel 1.
- INSTRUCTIONS: Número de instrucciones de CPU.
- L1-dcache-stores: Número de accesos de escritura a memoria caché nivel 1.
- CPU-CYCLES: Número de ciclos de CPU.
- CPU-CLOCK: Frecuencia de reloj del procesador.

Con estos contadores hardware hemos conseguido las siguientes métricas:

- Como métrica de acceso a memoria se utiliza los contadores “L1-dcache-loads” y “L1-dcache-stores” por instrucción (dividiendo esto por el número de instrucciones). Con esto obtenemos un valor normalizado sobre el que establecemos un umbral por encima del cual, el evento es de memoria.

- Como métrica de CPU se utiliza el número de instrucciones por ciclo y el número total de instrucciones, ya que si este valor es elevado, podemos entender que es un evento de CPU dado que estos realizan muchos cálculos aritméticos con el fin de someter a un estrés elevado a la CPU, aumentando a su vez las instrucciones por ciclo.

- Como métrica de I/O se ha utilizado el tiempo del sistema, ya que la frecuencia de reloj para los eventos de lectura y escritura en fichero es inferior a los eventos que realizan muchos ciclos de CPU o los eventos que acceden continuamente a memoria.

Estas fueron las métricas escogidas, ya que se realizaron varias pruebas monitorizando los distintos tipos de *benchmarks* utilizados, y eran capaces de diferenciar los tipos de procesos existentes.

El umbral escogido, a partir del cual el proceso se considera de un tipo o de otro, se ha determinado en base a las pruebas realizadas para escoger los eventos a utilizar. Estos umbrales se han definido realizando una media en base a diez pruebas que engloban la obtención del valor de todos los eventos para cada tipo de proceso en ejecución.

Los umbrales escogidos han sido los siguientes:

- Memoria: Si el valor obtenido por ambos contadores de acceso a memoria caché superan el valor ‘0.1’, este evento se categoriza como evento de memoria.

- CPU: Si el número de ciclos de CPU es superior a 6 billones y el número de instrucciones supera los 4 billones, este evento se considera un evento de CPU.

- I/O: Si la frecuencia de reloj es inferior a mil, y el número de accesos a memoria es inferior a 10 millones, el evento de considera de I/O.

Como es lógico, se podían haber escogido otro tipo de métricas, pero con los eventos que hemos sido capaces de capturar por limitaciones de la plataforma, estas métricas han sido suficientes para desarrollar nuestra aplicación.

4.1.1 Algoritmo de caracterización

El algoritmo utilizado para clasificar los tipos de procesos se ha generado en base a una batería de pruebas, donde se ha definido los umbrales, para que haya una clasificación correcta de los distintos tipos.

```
1.  tratarPeticion(){
2.          leerSocket(PID);
3.          leerSocket (contadores[] );
4.          leerSocket (prioridad);
5.          if(comprobarUmbralContadores(contadores[I/O])){
6.              bloquearMutex(mutexX);
7.              pararProceso(PID);
8.              añadirVector(PID,prioridad,vectorIO[][]);
9.              aumentarContador(punteroIO );
10.             liberarMutex(mutexX);
11.         }else if(comprobarUmbralContadores(contadores[CPU])){
12.             bloquearMutex (mutexY);
13.             pararProceso(PID);
14.             añadirVector(PID,prioridad,vectorCPU[][]);
15.             aumentarContador(punteroCPU);
16.             liberarMutex (mutexY);
17.         }else if(comprobarUmbralContadores(contadores[Memoria])){
18.             bloquearMutex (mutexZ);
19.             pararProceso(PID);
20.             añadirVector(PID,prioridad,vectorMEM[][]);
21.             aumentarContador(punteroMEM);
```

```
22.                 liberarMutex (mutexZ);
23.         }else{
24.                 terminarProceso(PID);
25.         }
```

Figura 8 - Algoritmo caracterización

A continuación, mostramos los comentarios asociados a la figura 8, señalando la línea donde va asociada el comentario.

Línea 2. Lectura PID proceso.
Línea 3. Lectura de los contadores hardware.
Línea 4. Lectura de la prioridad del proceso (si hay)
Línea 5. Ahora se evalúa el valor de los contadores leídos, para categorizar el proceso. Si los valores de los contadores que definen los procesos de I/O están dentro del umbral establecido, se evalúa como un evento de I/O. Si no, pasará a la siguiente condición para comprobar si es un proceso de otro tipo.
Línea 6. Se bloquea el mutex
Línea 7. Se detiene el proceso.
Línea 8. Se añade el PID y la prioridad del proceso al vector. Cuando se añade, se introduce en orden para evitar recorrer de nuevo el vector y empeorar el rendimiento del sistema.
Línea 9. Aumenta este contador, para apuntar a la siguiente posición del vector.
Línea 10. Se libera el mutex
Línea 11. Se evalúa el valor de los contadores para CPU
Línea 17. Se evalúa el valor de los contadores para Memoria
Línea 24. Si ha habido un error y no es capaz de categorizar el proceso, lo termina.

Normalmente, salvo que la lectura del contador haya sido errónea, la caracterización no tiene problemas para diferenciar el tipo de proceso porque los valores de los contadores son muy distintos según el tipo. Por este motivo, si no es capaz de caracterizar el tipo de proceso, lo detiene, evitando la ejecución de este en un momento erróneo, lo cual puede generar un aumento considerable en los tiempos de ejecución del planificador.

Por lo tanto, tendríamos 3 vectores que contienen todos los PID de los procesos, y ahora se pueden manejar directamente por el servidor, deteniendo o arrancando su ejecución.
Por otra parte, cada vez que se va a leer o escribir en los vectores, se bloquea el *mutex* del vector hasta que se termina la lectura/escritura, para evitar problemas de concurrencia. Además, los procesos del vector se reordenan por prioridad, de mayor a menor, ejecutándose por orden de llegada en caso de que tengan la misma prioridad, como se ha explicado en la sección 2.3 del presente documento.

Por el momento, la prioridad de los procesos no es tan importante como el hecho de clasificar el tipo de proceso (ya que el planificador principalmente se enfoca en la ejecución de procesos al

mismo tiempo en función del tipo, para mejorar los tiempos de ejecución). Por este motivo no se ha implementado un control de inanición, pero, lógicamente, para evitar que se produjera inanición, sería necesario incluir algún mecanismo de control, como por ejemplo, el aumento de la prioridad de un proceso después de un tiempo determinado.

4.1.2 Módulo planificador: Lógica para la generación de la planificación

Para la generación del planificador, se han utilizado distintas políticas, a partir de las cuales se han ido observando los resultados obtenidos para tratar de mejorarlo, hasta el punto de tener ganancias de tiempo muy grandes respecto al método de ejecución por defecto de la plataforma.

La lógica que se ha tratado de seguir ha sido siempre el acoplamiento de los distintos tipos de procesos en función de su compatibilidad, para que la ejecución conjunta de ambos no penalizara el tiempo total de ejecución de ambos al mismo tiempo. Las políticas utilizadas se muestran a continuación:

1. Política FIFO. Todos los procesos que llegan se van encolando y se ejecutan secuencialmente tratando de disminuir el tiempo gracias a la ganancia obtenida por no sobrecargar el procesador.

2. Política FIFO que ejecuta todos los procesos de un mismo tipo primero. Además, se aumenta el número de procesos en ejecución al mismo tiempo y se ejecutan siempre los procesos del mismo tipo juntos. Con esta política se busca explotar la ganancia que se obtiene sobrecargando el procesador de una manera mínima solapando procesos en ejecución.

3. Política FIFO que combina procesos de distintos tipos. Con esta política se explota la ganancia existente en que un proceso puede consumir más recursos de CPU mientras que otro que está en ejecución puede consumir más recursos de memoria, sobrecargando mínimamente el procesador ejecutando ambos los procesos al mismo tiempo.

4. Política FIFO con prioridad. Debido a los buenos resultados obtenidos con la política descrita en el punto 3, se decidió añadir prioridad a los procesos, ya que en una aplicación real, puede haber procesos más prioritarios que otros.

5. Política FIFO para todos los procesos excepto para los procesos de E/S, que se ejecutan todos juntos al llegar a un umbral determinado. De esta forma, se aprovecha que los procesos de E/S apenas generan un aumento de tiempo al ejecutarse a la vez, y se reduce la sobrecarga que se produciría si se ejecutan con otro tipo de procesos.

Como se puede observar, hay numerosas políticas, aunque todas están basadas en una política FIFO, ya que en una aplicación cliente-servidor, si no hay prioridad, el primer cliente que llega es el primero que recibe el servicio, si el sistema lo puede permitir.

Todas las políticas varían levemente, aunque esta variación se ha ido haciendo durante la experimentación, observando los resultados obtenidos, y pensando que se podía hacer para mejorarlos.

Todas estas políticas han tenido una batería de pruebas que mostremos en la sección 6.2.3. Prácticamente en la totalidad de los escenarios se ha salido siempre con ganancia de tiempo, debido a que el planificador por defecto ejecuta todos los procesos al mismo tiempo según llegan, de manera que al tener solamente un núcleo la plataforma, se genera una sobrecarga muy grande en el procesador. Estos experimentos se han hecho con una cantidad de procesos variable, pero con el planificador por defecto, sería prácticamente inviable sostener el sistema con una ejecución de 50 o más procesos, ya que el sistema no sería capaz de sostener la ejecución de todos al mismo tiempo.

4.1.3 Algoritmos

4.1.3.1 Algoritmo política 1

```
1.  tratar_proceso(){
2.      while(1){
3.          mientrasVectorProcesosNoVacio(){
4.              arrancarSiguienteProceso(vectorProcesos[]);
5.              esperaFinEjecucionProceso();
6.          }
7.      }
8.  }
```

Figura 9 - Algoritmo política 1

A continuación, mostramos los comentarios asociados a la figura 9, señalando la línea donde va asociada el comentario.

Línea 3. Si el vector contiene PID de procesos
Línea 4. Comienza la ejecución del siguiente proceso
Línea 5. Espera a que termine el proceso antes de que empiece la siguiente iteración

El planificador de la figura 9 no tiene prioridad para los procesos, por lo que el vector solo tiene una dimensión con el PID del proceso. Además, como los procesos no se diferencian en esta política por tipo, solamente hay un tipo de vector que contiene todos los PID de los procesos en orden de llegada.

4.1.3.2 Algoritmo política 2

```
1.  tratar_proceso_1(){
2.      while(1){
3.          if(comprobarContenidoVector(vectorX[])){
4.              bloquearMutex(mutexX);
5.              obtenerPID(vectorX[punteroX]);
6.              incrementarPuntero(punteroX);
7.              liberarMutex (mutexX);
8.              arrancarProceso(PID);
9.              esperarFinalizacionProceso();
10.         }else{
11.             if(comprobarContenidoVector(vectorY[])){
12.             }else{
13.                 if(comprobarContenidoVector(vectorZ[])){
14.                 }else{
15.                     sleep(15);
16.                 }
17.             }
18.         }
19.     }
20. }

21. tratar_proceso_2(){
22. }
```

Figura 10 - Algoritmo política 2

A continuación, mostramos los comentarios asociados a la figura 10, señalando la línea donde va asociada el comentario.

Línea 3. Comprueba si hay procesos de tipo X en espera
Línea 4. Se bloquea el mutex para proteger el acceso al PID del proceso.
Línea 5. Se obtiene el PID del proceso.
Línea 6. Incrementamos el contador para apuntar a la siguiente posición.
Línea 7. Se libera el mutex
Línea 8. Arranca la ejecución del proceso
Línea 9. Se espera a que este termine para realizar la siguiente iteración.
Línea 11. El código de esta parte es igual que el código para los procesos de tipo X anterior, sustituyendo las variables, que en este caso, pasarían a llamarse distinto (MutexY, vectorY, punteroY), y comprueba si hay procesos en espera de tipo Y.

Línea 13. El código de esta parte es igual que el código para los procesos de tipo X anterior, sustituyendo las variables, que en este caso, pasarían a llamarse distinto (MutexZ, vectorZ, punteroZ), y comprueba si hay procesos en espera de tipo Z.
Línea 15. No hay procesos en espera por lo que volvería a iterar a ver si han aparecido procesos nuevos, pero añadimos esto para que no ejecute en bucle iteraciones sobrecargando el sistema si no llegan nuevos procesos.
Línea 21. Contiene el mismo código que la función anterior, pero este proceso se encarga de ejecutar un segundo proceso de forma paralela, aumentando el número de procesos en ejecución.

Esta política que se muestra en la figura 10 ejecuta los procesos del mismo tipo hasta que no haya ninguno más en espera y pasa a ejecutar el siguiente tipo de proceso. El *mutex* bloquea el acceso a la variable que se incrementa para apuntar al siguiente PID de proceso dentro del vector.

De esta forma, se podrían priorizar los procesos del tipo que queramos, ya que solo habría que definir que procesos se añaden al vector X, y cuales al resto de vectores menos prioritarios.

*Nota: Se ha eliminado código duplicado para simplificar la comprensión del código. Se ha descrito en los comentarios el contenido de esas partes.

4.1.3.3 Algoritmo política 3

```
1.  tratar_proceso_1(){
2.      while(1){
3.          if(comprobarContenidoVector(vectorX[])){
4.              bloquearMutex(mutexX);
5.              obtenerPID(vectorX[punteroX]);
6.              incrementarPuntero(punteroX);
7.              liberarMutex(mutexX);
8.              arrancarProceso(PID);
9.              señalarVariableProceso(procesoX);
10.             esperarFinalizacionProceso();
11.             señalarProcesoFinalizado(procesoX);
12.         }else{
13.             if(comprobarContenidoVector(vectorY[])){
14.             }else{
15.                 if(comprobarContenidoVector(vectorZ[])
16.                 }else{
17.                     sleep(15);
18.                 }
19.             }
```

```
20.                     }
21.             }
22.     }
```

Figura 11 - Algoritmo política 3 (función 1)

A continuación, mostramos los comentarios asociados a la figura 11, señalando la línea donde va asociada el comentario.

Línea 3. Comprueba si hay procesos de tipo X en espera
Línea 4. Se bloquea el mutex para proteger el acceso al PID del proceso.
Línea 5. Se obtiene el PID del proceso.
Línea 6. Incrementamos el contador para apuntar a la siguiente posición.
Línea 7. Se libera el mutex
Línea 8. Arranca la ejecución del proceso
Línea 9. Cambia el valor de la variable para que el segundo hilo que ejecuta los procesos sepa qué tipo de proceso hay en ejecución.
Línea 10. Se espera a que el proceso en ejecución termine para realizar la siguiente iteración.
Línea 11. El proceso X ha terminado, la variable vuelve a su valor por defecto.
Línea 13. El código de esta parte es el mismo que el código para los procesos de tipo X anterior, sustituyendo las variables. Ahora pasarían a llamarse de distinta forma (MutexY, vectorY, punteroY, procesoY). Además, la comprobación la realiza sobre el vector de procesos Y.
Línea 15. El código de esta parte es el mismo que el código para los procesos de tipo X anterior, sustituyendo las variables. Ahora pasarían a llamarse de distinta forma (MutexZ, vectorZ, punteroZ, procesoZ). Además, la comprobación la realiza sobre el vector de procesos Z.
Línea 17. No hay procesos en espera por lo volvería a iterar a ver si han aparecido procesos nuevos.

```
1.      tratar_proceso_2(){
2.              while(1){
3.                      if(procesoXEjecutando()){
4.                              if(comprobarContenidoVector(vectorY[])){
5.                                      bloquearMutex(mutexY);
6.                                      obtenerPID(vectorY[punteroY]);
7.                                      incrementarPuntero(punteroY);
8.                                      liberarMutex (mutexY);
9.                                      arrancarProceso(PID);
10.                                     esperarFinalizacionProceso();
11.                             }else if(comprobarContenidoVector(vectorZ[])){
12.                             }else if(comprobarContenidoVector(vectorX[])){
13.                             }
14.                     }else if(procesoYEjecutando()){
```

```
15.                         if(comprobarContenidoVector(vectorX[])){
16.                             bloquearMutex(mutexX);
17.                             obtenerPID(vectorX[punteroX]);
18.                             incrementarPuntero(punteroX);
19.                             liberarMutex (mutexX);
20.                             arrancarProceso(PID);
21.                             esperarFinalizacionProceso();
22.                         }else if(comprobarContenidoVector(vectorZ[])){
23.                         }else if(comprobarContenidoVector(vectorY[])){
24.                         }
25.                     }else if(procesoZEjecutando()){
26.                         if(comprobarContenidoVector(vectorX[])){
27.                             bloquearMutex(mutexX);
28.                             obtenerPID(vectorX[punteroX]);
29.                             incrementarPuntero(punteroX);
30.                             liberarMutex (mutexX);
31.                             arrancarProceso(PID);
32.                             esperarFinalizacionProceso();
33.                         }else if(comprobarContenidoVector(vectorY[])){
34.                         }else if(comprobarContenidoVector(vectorZ[])){
35.                         }else{
36.                             sleep(15);
37.                         }
38.                     }
39.                 }
40.     }
```

Figura 12 - Algoritmo política 3 (función 2)

A continuación, mostramos los comentarios asociados a la figura 12, señalando la línea donde va asociada el comentario.

Línea 3. Si el primer hilo está ejecutando un proceso de tipo X, ejecutara un proceso del resto de tipos si es posible, para no combinar dos procesos del mismo tipo.
Línea 4. Comprueba si hay procesos de tipo Y en espera.
Línea 5. Se bloquea el mutex para proteger el acceso al PID del proceso.
Línea 6. Se obtiene el PID del proceso.
Línea 7. Incrementamos el contador para apuntar a la siguiente posición.
Línea 8. Se libera el mutex
Línea 9. Arranca la ejecución del proceso
Línea 10. Se espera a que este termine para realizar la siguiente iteración.

Línea 11. Realiza la misma comprobación de antes, pero esta vez al vector que contiene los procesos de tipo Z. El flujo si entra en la condición sería el mismo explicado en los comentarios anteriores, con las variables asociadas a Z.
Línea 12. Realiza la misma comprobación de antes, pero esta vez al vector que contiene los procesos de tipo X. El flujo si entra en la condición sería el mismo explicado en los comentarios anteriores, con las variables asociadas a X.
Línea 14. Comprueba si el primer hilo está ejecutando un proceso de tipo Y. Si es así, realiza las mismas operaciones comentadas anteriormente desde la línea 3 a la 12, cambiando el orden de procesos para tartar de no ejecutar dos procesos de tipo Y al mismo tiempo (respetando la prioridad de X sobre Z).
Línea 25. Comprueba si el primer hilo está ejecutando un proceso de tipo Z. Si es así, realiza las mismas operaciones comentadas anteriormente desde la línea 3 a la 12, cambiando el orden de procesos para tartar de no ejecutar dos procesos de tipo Z al mismo tiempo (respetando la prioridad de X sobre Y).
Línea 36. No hay procesos en espera, esperamos 15 segundos para ejecutar la siguiente iteración para no sobrecargar el sistema.

Para la política mostrada en la figura 12, se definirá la ejecución de qué procesos es prioritaria, de manera que solamente habría que decidir cual queremos que se ejecute en primer lugar y cual en último lugar. En el código, 'X' sería el más prioritario y 'Z' el menos.

El primer hilo siempre ejecutará los procesos vaciando el vector del tipo más prioritario, mientras que el segundo hilo, ejecutará otro tipo de proceso distinto al que ya se está ejecutando.

De esta manera, normalmente se ejecutarán procesos de distinto tipo, obteniendo la ganancia que se produce al no sobrecargar el mismo recurso del sistema con ambos procesos, obteniendo una mejora del rendimiento.

4.1.3.4 Algoritmo política 4

Como ya se comentó en la sección 4.1.2, en la definición de la política 4, esta política solo se diferencia de la anterior en que se añade prioridad a los procesos. Por lo tanto, el único cambio existente respecto al pseudocódigo anterior es que los vectores tienen una dimensión más (donde se encontraría la prioridad del proceso).

Sin embargo, para evitar problemas de rendimiento (en caso de que hubiera muchos procesos, o el tamaño del vector llegara a crecer mucho), la prioridad de los procesos se tiene en cuenta a la hora de añadirlos en el vector. De esta manera, solo recorreremos el vector una vez, añadiendo el proceso que acaba de llegar al vector de manera ordenada en función de su prioridad. Así solo habría que coger el siguiente elemento del vector para empezar su ejecución, ya que este sería más prioritario que el siguiente.

Con esta decisión, evitamos recorrer el vector para buscar que proceso es el más prioritario, y así no se harán iteraciones redundantes, mejorando el rendimiento de la aplicación.

Por último, para ser fiel a la política FIFO, los procesos se encolan por orden de llegada, de manera que un proceso con la misma prioridad que ha llegado después, se añadirá después en el vector.

Por otra parte, la inanición no se baraja, dado que estamos haciendo un análisis de rendimiento, y no es importante si un proceso se ejecuta antes o después, pero sería fácil solucionarlo, ya que se podría aumentar la prioridad de los procesos, si al cabo de un número determinado de ejecuciones este no ha llegado a ejecutarse.

4.1.3.5 Algoritmo política 5

Esta política solo se diferencia de la política anterior en el segundo hilo encargado de tratar los distintos tipos de procesos distintos al primer hilo.

Como ya comentamos en la 'política 3' de la sección 4.1.3.3, el primer hilo será el encargado de ejecutar el tipo de proceso más prioritario (que en este caso sería CPU o memoria porque en esta política los procesos de E/S se tratan de una manera distinta para mejorar el rendimiento de la aplicación). Mientras tanto, el segundo hilo será el encargado de comprobar si el número de procesos de E/S en espera supera un umbral determinado.

Cuando se ha llegado al umbral, se ejecutan todos los procesos de E/S de una sola vez, mientras que si no se ha llegado, la ejecución se realiza de manera normal, como en las políticas anteriores.

El nuevo hilo en esta política añadiría el siguiente fragmento mostrado en la figura 13 para comprobar si se ha llegado al umbral de procesos de E/S:

```
1.  Tratar_proceso_2(){
2.      While(1){
3.          If(comprobarUmbral(procesosEsperaIO)){
4.              ejecutarProcesosIOEspera();
5.              esperarEjecucionProcesosIO();
6.              resetearContadorProcesosIO();
7.          }else{
8.          }
9.      }
10. }
```

Figura 13 - Algoritmo política 5

A continuación, mostramos los comentarios asociados a la figura 13, señalando la línea donde va asociada el comentario.

Línea 3. Comprueba si el número de procesos de E/S en espera ha llegado al umbral determinado.
Línea 4. Se ejecutan todos los procesos de IO en espera al mismo tiempo.
Línea 5. Se espera a que termine el último.
Línea 6. Se resetea el contador con el numero de procesos de IO en espera.
Línea 7. Si el número de procesos de E/S en espera no llega al umbral, seguiría el mismo flujo mostrado en la figura 9 de la política 3.

La variable '*procesosEsperaIO'* se va actualizando conforme el proceso '*tratar_peticion()*' de la sección 4.1.1 va insertando nuevos procesos de I/O en el vector asociado.

Por otra parte, ya que esta función ejecutaría en bloques los procesos de I/O, la otra función encargada de ejecutar procesos solo ejecutará procesos de CPU y memoria.

Estas serían todas las políticas implementadas para nuestro planificador. Las variables/vectores que se utilizan, como son leídas y escritas por distintos hilos, son variables/vectores globales (como las variables que realizan la función de punteros con el índice de los vectores y los propios vectores que contienen los PID de los procesos) que se protegen con un *mutex* para evitar que haya problemas con las condiciones de carrera en el momento de leerlo/escribirlo.

Por último, para dar una visión global de donde se ejecutan los algoritmos, se muestra la figura 14.

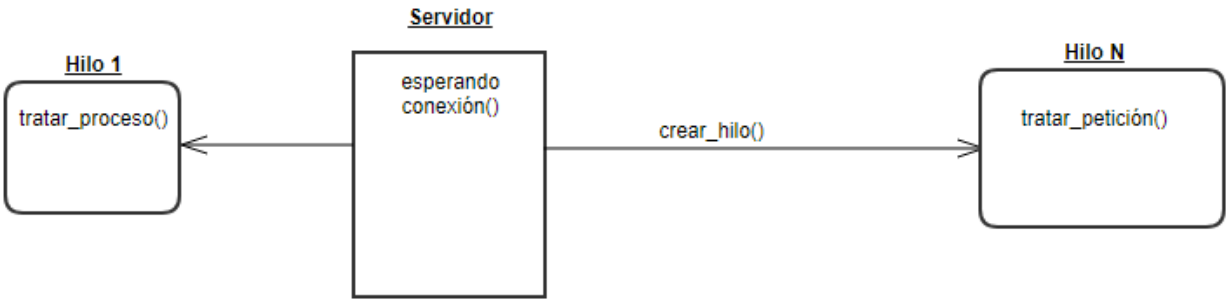


Figura 14 - Visión global servidor

La primera vez que el servidor recibe una conexión entrante, crea el 'Hilo 1', que será el encargado de ejecutar los algoritmos para monitorizar los procesos. Hace la función del planificador.

Por otra parte, cada vez que haya una conexión entrante, el servidor creará un hilo independiente encargado de manejar la petición entrante, liberando de esta manera al servidor.

4.2 Conexión entre los componentes: protocolo de comunicación y señales

La comunicación que se realiza entre el cliente y el proceso, no es una comunicación directa entre ambos, ya que, como los clientes no saben que procesos van a llegar, el PID del proceso que van a monitorizar lo leen de un fichero donde los procesos han añadido su PID según van llegando. Por lo tanto, las señales utilizadas serían las funciones *open(), read(), write() y close()*. Es decir, señales para el manejo de ficheros.

Por otra parte, sí existe comunicación directa entre el cliente y el servidor. Esta es una comunicación TCP a través de sockets. Las señales utilizadas durante esta comunicación son las siguientes:

- connect(): función para que el cliente establezca la conexión con el servidor.
- accept(): el servidor se queda en espera para aceptar conexiones entrantes.
- read(): se lee el valor enviado por la otra parte de la conexión a través del socket, una vez se haya establecido la conexión.
- write(): se escribe el valor que se quiere enviar a través de socket una vez se haya establecido la conexión.
- close(): se cierra la conexión establecida a través del socket.

Por último, entre el servidor y el proceso también se realiza una comunicación a través de distintas señales, como son las siguientes:

- kill (PID,SIGSTOP): Esta señal detiene la ejecución del proceso asociado al PID, y lo pone en espera.
- kill(PID,SIGCONT): Esta señal reanuda la ejecución del proceso asociado al PID.
- kill(PID,0): Esta señal nos devolverá el valor 0 si el proceso asociado al PID existe. De esta manera, sabremos si el proceso ha terminado su ejecución.

Estas son todas las señales utilizadas para el desarrollo de nuestro planificador durante la comunicación existente entre todas las partes de la aplicación.

Capítulo 5: Análisis

5.1 Especificación de requisitos

En la tabla 1 mostraremos la plantilla que se utilizará para la especificación de requisitos:

Identificador: Tipo-XX	
Título	
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	

Tabla 1 - Plantilla de requisitos

Los campos que se recogen son los siguientes:

- Titulo: título representativo del requisito.
- Prioridad: indica la necesidad que tiene de implementación en la fase de construcción de la aplicación. La prioridad alta implica que el requisito tiene que implementarse al inicio del desarrollo de la aplicación.
- Necesidad: grado de importancia del requisito.
- Descripción: explicación del requisito.

5.1.1 Requisitos de capacidad

Representan las funciones y operaciones que el sistema debe ser capaz de realizar.

Identificador: CAP-01	
Título	Monitorizar procesos
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema será capaz de monitorizar procesos entrantes.

Tabla 2 - Requisito CAP-01

Identificador: CAP-02	
Título	Monitorizar contadores hardware
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema será capaz de obtener el valor los contadores hardware asociados a un proceso entrante.

Tabla 3 - Requisito CAP-02

Identificador: CAP-03	
Título	Generador clientes
Prioridad	☐ Alta ☐ Media ☒Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema será capaz de crear nuevos clientes para tratar los procesos entrantes.

Tabla 4 - Requisito CAP-03

Identificador: CAP-04	
Título	N clientes y procesos
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema tendrá tantos clientes como procesos entrantes.

Tabla 5 - Requisito CAP-04

Identificador: CAP-05	
Título	Servidor en recepción
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema tendrá un servidor capaz de aceptar las conexiones entrantes por parte de clientes.

Tabla 6 - Requisito CAP-05

Identificador: CAP-06	
Título	Comunicación cliente-servidor
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema tendrá un numero N de cliente capaces de establecer la conexión con un servidor.

Tabla 7 - Requisito CAP-06

Identificador: CAP-07	
Título	Manejador ficheros
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema será capaz de abrir, escribir, leer y cerrar ficheros.

Tabla 8 - Requisito CAP-07

Identificador: CAP-08	
Título	Servidor generador.
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor del sistema será capaz de crear hilos independientes.

Tabla 9 - Requisito CAP-08

Identificador: CAP-09	
Título	Servidor único
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema tendrá un único servidor.

Tabla 10 - Requisito CAP-09

5.1.2 Requisitos de restricción

Estos requisitos representan aquellas limitaciones de nuestra aplicación, así como las incompatibilidades de la misma.

Identificador: RES-01	
Título	SSOO
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El sistema operará con un sistema operativo UNIX.

Tabla 11 - Requisito RES-01

Identificador: RES-02	
Título	Conexión proceso-servidor
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los procesos del sistema no podrán establecer conexión directa con el servidor.

Tabla 12 - Requisito RES-02

Identificador: RES-03	
Título	Flujo constante
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El proceso de monitorización de los procesos no alterará el funcionamiento de la aplicación.

Tabla 13 - Requisito RES-03

Identificador: RES-04	
Título	Fichero con PID
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El cliente obtendrá el PID de los procesos entrantes de un archivo predeterminado.

Tabla 14 - Requisito RES-04

Identificador: RES-05	
Título	Proceso exclusivo por cliente
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Un cliente no podrá monitorizar más de un proceso.

Tabla 15 - Requisito RES-05

Identificador: RES-06	
Título	Cliente exclusivo por proceso
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Un proceso no podrá ser monitorizado por más de un cliente.

Tabla 16 - Requisito RES-06

Identificador: RES-07	
Título	Tipos de procesos
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los procesos solo pueden ser de tres tipos: “CPU, Memoria y E/S”

Tabla 17 - Requisito RES-07

Identificador: RES-08	
Título	Compatibilidad servidor.
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los servidor solo será capaz de monitorizar procesos de tres tipos: “CPU, Memoria y E/S”

Tabla 18 - Requisito RES-08

Identificador: RES-09	
Título	Tratamiento de procesos sin tipificar
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor no monitorizará ningún proceso que no sea capaz de tipificar.

Tabla 19 - Requisito RES-09

Identificador: RES-10	
Título	Detención procesos
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor será el único capaz de detener procesos.

Tabla 20 - Requisito RES-10

Identificador: RES-11	
Título	Arrancar procesos
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor será el único capaz de arrancar procesos.

Tabla 21 - Requisito RES-11

5.1.3 Requisitos funcionales

Estos requisitos representan lo que la aplicación es capaz de proporcionarnos, así como la funcionalidad que debe seguir la aplicación desarrollada.

Identificador: FUN-01	
Título	Método monitorizar procesos
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los clientes monitorizaran los procesos en función de su PID.

Tabla 22 - Requisito FUN-01

Identificador: FUN-02	
Título	Contadores hardware compatibles.
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los clientes serán capaces de obtener el valor de los siguientes contadores hardware: “L1-dcache-loads, Instructions, L1-dcache-stores, CPU-Cycles, CPU-Clock”.

Tabla 23 - Requisito FUN-02

Identificador: FUN-03	
Título	Envío datos cliente-servidor.
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los clientes enviarán los datos de los contadores hardware al servidor.

Tabla 24 - Requisito FUN-03

Identificador: FUN-04	
Título	Herramienta Perf
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los clientes utilizaran la herramienta *Perf* para monitorizar los procesos.

Tabla 25 - Requisito FUN-04

Identificador: FUN-05	
Título	Benchmark exclusivo por proceso.
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Cada proceso ejecutará un único *benchmark*.

Tabla 26 - Requisito FUN-05

Identificador: FUN-06	
Título	Finalización cliente
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Cada cliente parará su ejecución una vez haya enviado todos los datos al servidor.

Tabla 27 - Requisito FUN-06

Identificador: FUN-07	
Título	Hilo tratamiento petición
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor creará un hilo independiente para el tratamiento de cada cliente entrante.

Tabla 28 - Requisito FUN-07

Identificador: FUN-08	
Título	Hilo planificador
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor creará un hilo encargado de realizar el funcionamiento del planificador.

Tabla 29 - Requisito FUN-08

Identificador: FUN-09	
Título	Detención proceso tratado
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor detendrá todo proceso entrante nada mas clasificarlo.

Tabla 30 - Requisito FUN-09

Identificador: FUN-10	
Título	Arranque mediante hilo planificador.
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los procesos serán arrancados por el hilo encargado de realizar las funciones del planificador.

Tabla 31 - Requisito FUN-10

Identificador: FUN-11	
Título	Detención proceso sin tipificar.
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor detendrá aquellos procesos que no sea capaz de tipificar.

Tabla 32 - Requisito FUN-11

Identificador: FUN-12	
Título	Finalización hilos tratamiento petición.
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los hilos creados por el servidor para tratar las peticiones de los clientes se destruirán una vez finalizada su ejecución.

Tabla 33 - Requisito FUN-12

Identificador: FUN-13	
Título	Servidor en espera.
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor siempre estará esperando nuevas conexiones a través del socket.

Tabla 34 - Requisito FUN-13

Identificador: FUN-14	
Título	Finalización procesos.
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Los procesos finalizaran una vez ejecutado su *benchmark* correspondiente.

Tabla 35 - Requisito FUN-14

Identificador: FUN-15	
Título	Protección condiciones de carrera.
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El acceso de lectura/escritura a los vectores se protegerá con *mutex*.

Tabla 36 - Requisito FUN-15

Identificador: FUN-16	
Título	Borrado PID leidos.
Prioridad	☐ Alta ☒ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El PID de los procesos leídos por parte de los clientes serán eliminados del fichero que los contiene.

Tabla 37 - Requisito FUN-16

Identificador: FUN-17	
Título	Conexión TCP
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	La conexión cliente-servidor será establecida a través de sockets TCP.

Tabla 38 - Requisito FUN-17

Identificador: FUN-18	
Título	Conexión señales
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El servidor se comunicará con los procesos a través de señales.

Tabla 39 - Requisito FUN-18

5.1.4 Requisitos no funcionales

Representan aquellas características requeridas del sistema, del proceso de desarrollo, del servicio prestado o de cualquier otro aspecto del desarrollo, que señala una restricción del mismo.

Identificador: NOF-01	
Título	Conexión a la red.
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☐ Alta ☐ Media ☒ Baja
Descripción	La plataforma no necesitará conexión a Internet.

Tabla 40 - Requisito NOF-01

Identificador: NOF-02	
Título	Raspberry Pi
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	El entorno de funcionamiento de la aplicación será una Raspberry Pi.

Tabla 41 - Requisito NOF-02

Identificador: NOF-03	
Título	Visualización aplicación.
Prioridad	☐ Alta ☐ Media ☒ Baja
Necesidad	☐ Alta ☐ Media ☒ Baja
Descripción	Para la visualización de la aplicación será necesario un monitor con conexión HDMI.

Tabla 42 - Requisito NOF-03

Identificador: NOF-04	
Título	Integridad información.
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	La información debe llegar de forma íntegra y ordenada al servidor.

Tabla 43 - Requisito NOF-04

Identificador: NOF-05	
Título	Almacenamiento Raspberry Pi
Prioridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Alta ☐ Media ☐ Baja
Descripción	Será necesaria una tarjeta SD para el almacenamiento en la Raspberry Pi.

Tabla 44 - Requisito NOF-05

5.2 Definición de interfaces de usuario: datos de entrada y salida

A continuación vamos a mostrar el flujo que se produce en la aplicación desde la llegada de un proceso entrante hasta que finaliza su ejecución en la aplicación. Como el flujo es el mismo, independientemente del tipo de proceso, no hará falta mostrar el flujo por cada tipo de proceso existente en la aplicación:

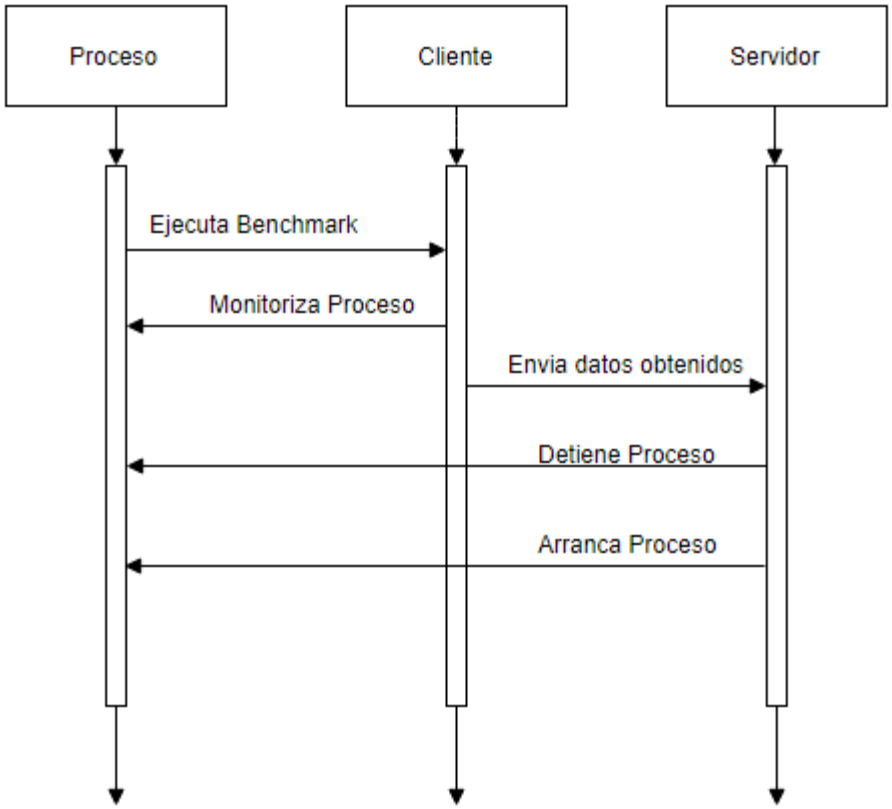


Figura 15 - Flujo datos entrada/salida

La figura 15 muestra la secuencia de ejecución normal, cuando el servidor ha sido capaz de tipificar el proceso. Por otra parte, se muestra un caso de uso poco probable como es el expuesto en la figura 16, ya que muestra el flujo que se produciría si el servidor no fuera capaz de tipificar el proceso.

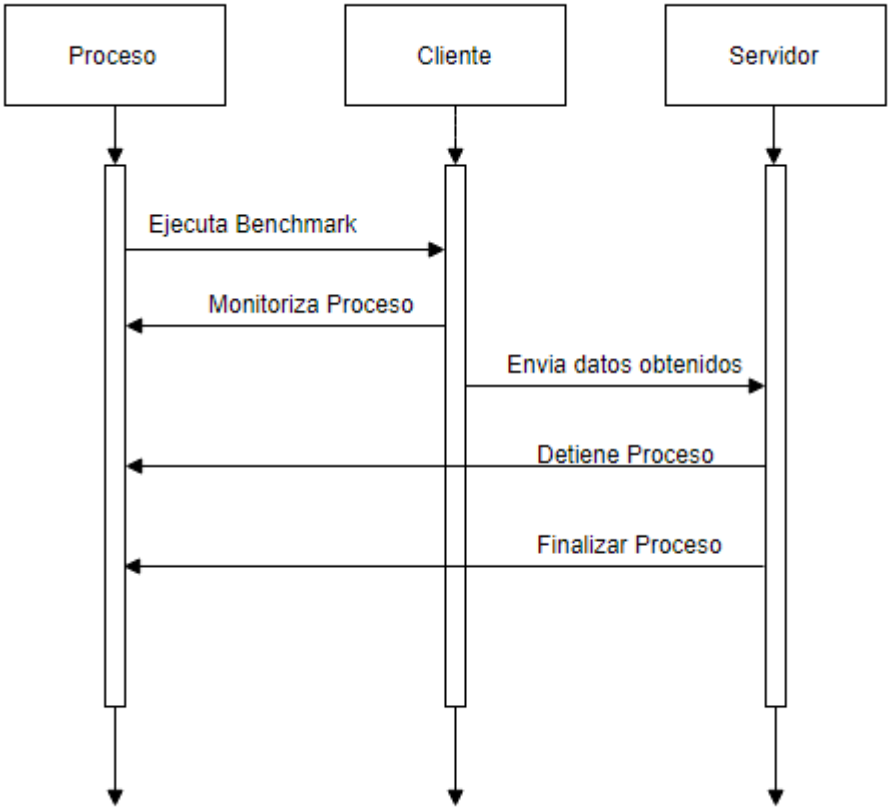


Figura 16 - Flujo entrada/salida si error

Con esto terminaríamos los casos de uso posibles, ya que el flujo de la aplicación no varía en función del proceso.

Capítulo 6: Evaluación y pruebas

6.1 Benchmarks utilizados

Como ya se explico en la sección 3.2.2 del documento, se han utilizado tres tipos de benchmarks para el desarrollo de nuestra aplicación. Lógicamente, para que la ejecución de los mismos sea posible, será necesario haber instalado previamente las librerías asociadas de estos benchmarks.

- Benchmark de CPU: Se encarga de someter al máximo estrés posible la CPU, de manera que esta será el recurso más demandado cuando esté en ejecución. El comando para ejecutarlo ha sido el siguiente:

  ./sysbench --test=cpu run

- Benchmark de Memoria: Genera un gran número de llamadas a memoria, de manera que genera un estrés mayor en esta parte. El comando para ejecutarlo ha sido el siguiente:

  ./sysbench --test=memory run

- Benchmark de E/S: Genera un mayor número de llamadas de entrada y salida, por lo que somete a un estrés mayor a esta parte del sistema. El comando para ejecutarlo es el siguiente:

  ./sysbench --test=fileio --file-total-size=4G --file-num=64 --file-test-mode=rndwr run

En todos estos benchmarks, se puede añadir distintos *flags* para cambiar la ejecución del benchmark, aunque el tipo de recurso que consumirá el mismo no cambiará. En nuestro caso, nos ha sido suficiente con los *flags* determinados en las llamadas anteriores, aunque se podrían añadir otros también.

6.2 Diseño de pruebas

6.2.1 Especificación de pruebas

Las pruebas realizadas para probar el funcionamiento del planificador han sido numerosas. Estas pruebas a su vez, en función de los resultados obtenidos, nos han ayudado a perfeccionar el planificador basándonos en el análisis de los resultados y el comportamiento de los mismos en función de los tipos de *benchmarks* ejecutados.

Todas las pruebas se han ejecutado en baterías de 10, de manera que los resultados son la media de los valores obtenidos. De esta manera, seríamos capaces de evitar valores inesperados debido a comportamientos inesperados, además de hacer más fiable el resultado obtenido.

Las pruebas han consistido en la ejecución de un número determinado de *benchmarks*, de distintos o del mismo tipo, usando el planificador desarrollado y usando el planificador por defecto. De esta manera, hemos sido capaces de observar la mejora de rendimiento producida por nuestro planificador con respecto al planificador por defecto que hay en el sistema.

Pruebas

1. Ejecución de distintos *benchmarks* con planificador y sin él.

2. Ejecución de un máximo de dos tipos de *benchmarks* con planificador y sin él, solapando *benchmarks* de distintos tipos durante la ejecución en el orden de 1 + 1 (solamente se ejecutarán un máximo de 2 procesos al mismo tiempo, siendo estos de distintos tipos si es posible).

3. Ejecución de todos los tipos de *benchmarks* en conjuntos de 10, 12 y 16 procesos, para someter a una sobrecarga mayor al procesador sin solapar la ejecución de procesos durante la ejecución.

4. Ejecución de distintos tipos de *benchmarks* en conjuntos de 10, 12 y 16 procesos, para someter a una sobrecarga mayor al procesador solapando la ejecución de procesos durante la ejecución.

5. Ejecución de distintos tipos de *benchmarks*, ejecutando los de E/S conjuntamente cuando llegan a un umbral determinado estando en espera y solapando la ejecución del resto.

6. Ejecución inicial todos los *benchmarks* de E/S (suponiendo un escenario ideal donde conocemos los entrantes) y solapando el resto durante la ejecución.

7. Ejecución de los *benchmarks* otorgándoles un valor de prioridad para probar el modelo desarrollado que utiliza procesos prioritarios.

Todas estas pruebas generadas se presentarán siguiendo el siguiente formato:

- Escenario Prueba-X: Identificador que representa el número de la prueba.

- Procesos iniciales y resultados obtenidos: Se muestran en una tabla los distintos tipos de *benchmarks* que van a intervenir en la prueba, además de los resultados obtenidos durante la prueba.

- Grafica: Gráfica comparativa entre el resultado con el planificador y el resultado sin él.

- Ganancia/Perdida total: Será la suma total del tiempo que se ha ganado o perdido en esta prueba respecto al uso del planificador.

6.2.2 Validación de modelado

Teniendo en cuenta que nuestra aplicación está basada en el consumo de recursos, nuestra validación para que el modelo sea correcto tendrá en cuenta el rendimiento del sistema con el uso de nuestro planificador.

Por el motivo expuesto, para que la validación del modelo sea correcta, tendremos que mejorar el rendimiento con el uso del planificador. Por lo tanto, vamos a establecer un umbral de mejora del 60% de los escenarios posibles para dar por bueno el modelo.

Para todo esto, tendremos en cuenta todos los casos de prueba realizados, que se asemejan a un posible escenario real, aunque en este podría darse el caso de que llegaran procesos erróneos al servidor, ataques de DoS… con tal de empeorar el rendimiento de nuestra aplicación. Pero estos casos no los vamos a tener en cuenta para las pruebas, ya que para esto hay otro tipo de métodos que no interfieren en el objetivo de nuestro proyecto.

Por este motivo, tendremos en cuenta toda la batería de pruebas que se ha utilizado de manera local, planteando diferentes y numerosos escenarios, para poder simular de la mejor manera posible un escenario real.

Por último, cabe destacar que para que nuestro modelo sea satisfactorio, hay que tener en cuenta el posible rendimiento de la aplicación con el aumento de procesos, o mayor demanda para el servidor, comparando si en estos casos se produciría una mejora del rendimiento o si empeoraría.

Esta validación finalmente se dará por satisfactoria o no cuando se realice un análisis de los resultados obtenidos, en la sección 6.2.4 del documento.

6.2.3 Batería de Pruebas

6.2.3.1 Escenario Prueba - 01

Procesos iniciales y resultados obtenidos

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/Perdida (s)
CPU	485	485	0
CPU / CPU	976	485 / 971	-480
CPU / CPU / CPU	1464	494 / 989 / 1483	-1502
MEM	901	898	3
MEM / MEM	1731	930 / 1810	-1009
MEM / MEM / MEM	2785	999 / 1918 / 2845	-2977
IO	352	488	-136
IO / IO	522	486 / 971	-935
IO / IO / IO	1611	499 / 996 / 1488	-1372
CPU / IO	495 / 488	486 / 961	-464
CPU / MEM	968 / 1286	491 / 1414	349
CPU / MEM / IO	991 / 1482 / 483	493 / 1385 / 1874	-796
CPU / CPU / MEM	1466 / 1466 / 1845	495 / 989 / 1845	1448
MEM / IO / IO	911 / 527 / 527	881 / 1327 / 1816	-2059
MEM / MEM / CPU	2204 / 2243 / 1464	1397 / 2308 / 496	1710
MEM / IO	960 / 469	955 / 1441	-967

Tabla 45 - Tiempos prueba 01

*Nota: Cuando se utiliza el planificador por defecto, los benchmarks ejecutados del mismo tipo acaban al mismo tiempo, mientras que cuando ejecutas de distinto tipo, acaban a distinto tiempo. Por este motivo, se representa el tiempo de ejecución de cada benchmark separado por '/' cuando son de distinto tipo o cuando son ejecutados por el planificador desarrollado, que los ejecuta serialmente.

Grafica

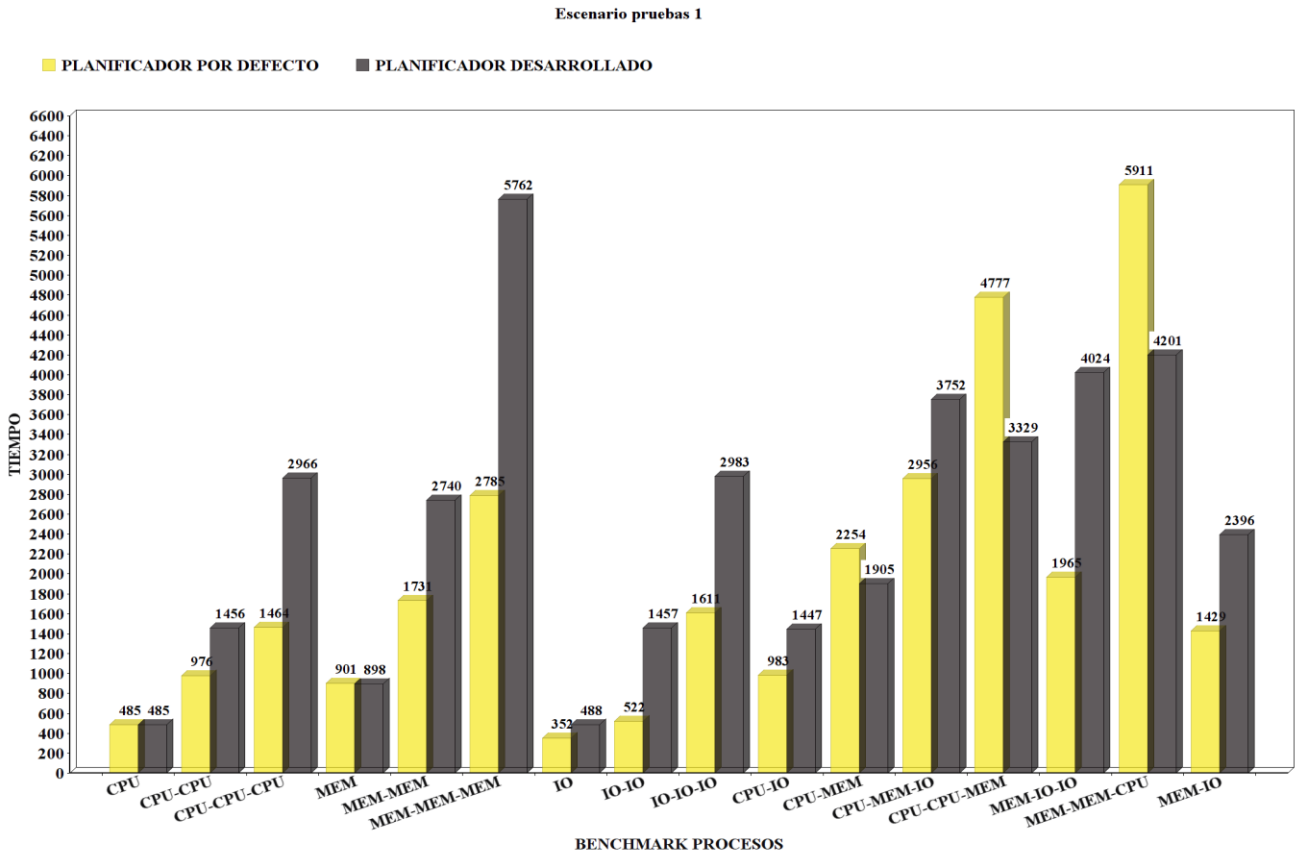


Figura 17 - Gráfica prueba 01

Ganancia/Perdida total

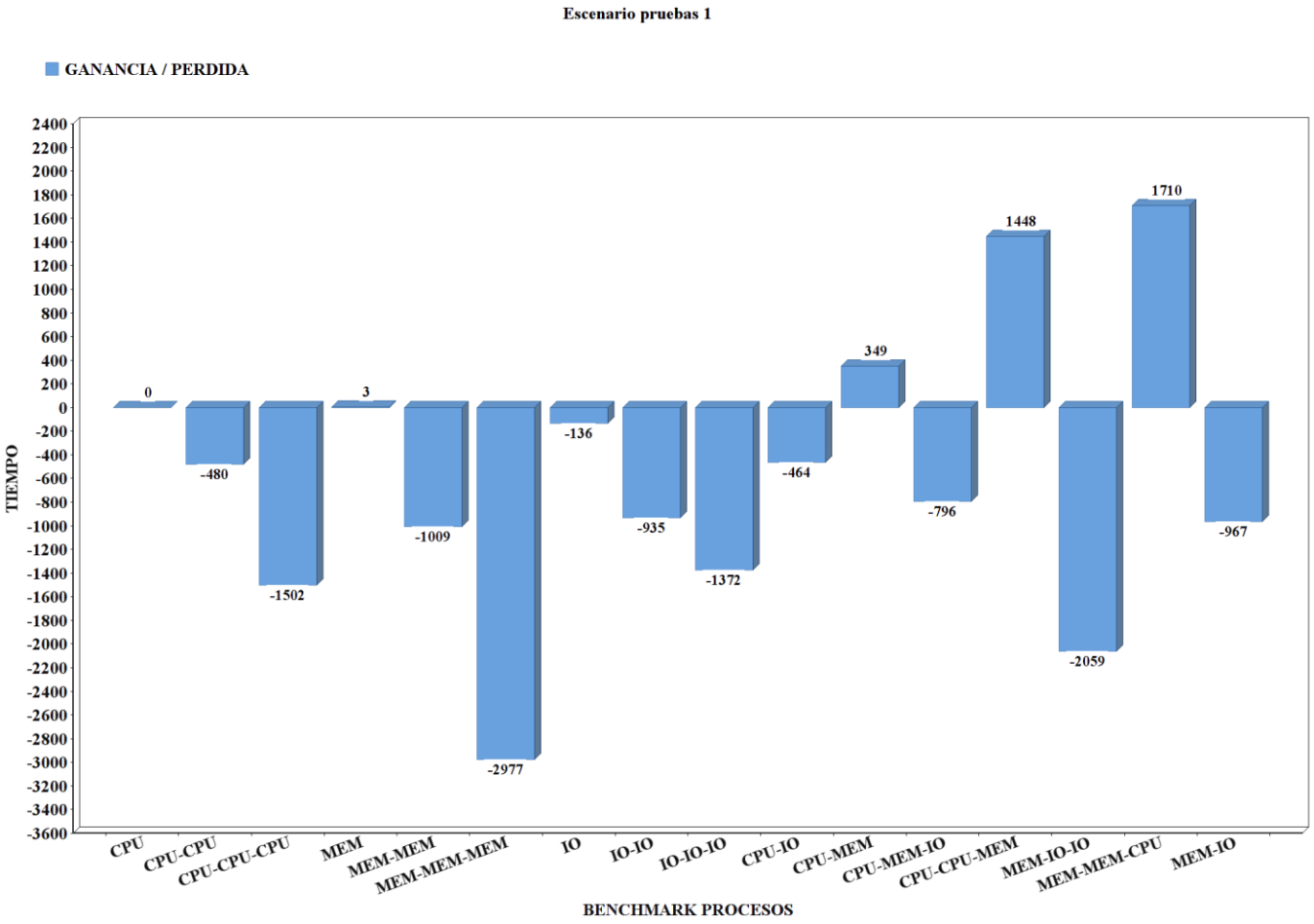


Figura 18 - Grafica ganancia/pérdida prueba 01

Como podemos observar en las figuras 17 y 18, la ejecución con un número de procesos pequeño no genera una ganancia destacable, ya que en la mayoría de los casos, se genera una pérdida de rendimiento, porque la suma del tiempo empleado por cada proceso para ejecutarse es mayor con el planificador desarrollado.

Lo que si se observa, es que combinando los distintos tipos de benchmarks, se genera una pequeña mejora respecto a los casos en los que se ejecutan del mismo tipo. Esto se puede deber a que la ejecución de distintos tipos de *benchmarks* al mismo tiempo genera una mayor sobrecarga al procesador.

Este caso de prueba nos lleva a generar el siguiente escenario, en el cual aumentamos la sobrecarga del procesador incluyendo más procesos, además de mezclar distintos tipos de los mismos para simular escenarios más reales.

6.2.3.2 Escenario Prueba - 02

Procesos iniciales y resultados obtenidos

En la tabla 46 que se muestra a continuación, el formato que se ha seguido al mostrar los tiempos representa el tiempo total que tarda en ejecutarse cada proceso separado por '/'. Por ejemplo, para el primer caso (3 procesos de CPU y 3 de memoria), cada proceso de CPU tardó 2920, 2922 y 2922 segundos respectivamente, mientras que cada proceso de memoria tardó 4136, 4155 y 4184 segundos.

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/Perdida (s)
3 procesos CPU + 3 procesos memoria	2920/2922/2922 + 4136/4155/4184	989/2350/3769 + 1367/2802/4195	5767
2 proceso de CPU + 2 procesos de memoria	1940/1942 + 2743/2750	983/2376 + 1388/2793	1835
4 procesos de CPU + 4 procesos de memoria	3879/3887/3888/3886+ 5462/5323/5313/5434	995/2389/3710/5119 + 1398/2735/4158/5500	11068
4 proceso de CPU + 4 procesos de IO	1969/1970/1969/1967 + 899/900/899/897	553/1042/1539/2049 + 503/1026/1532/2034	1192
4 procesos de IO + 4 procesos de memoria	1009/1012/1006/998+ 3647/3665/3717/3729	452/1380/2240/3188 + 898/1751/2696/3577	2601
4 procesos de CPU + 4 procesos de CPU	3887/3898/3894/3893/ 3892/3894/3889/3891	998/997/1967/1971/ 2943/2943/3916/3917	11486
4 proceso de IO + 4 procesos de IO	592/591/590/589/ 588/587/586/584	498/500/1506/1508/ 2096/2098/2712/2713	-8924
4 procesos de memoria + 4 procesos de memoria	7012/7037/6847/7035/ 7050/7036/6743/6984	2612/2623/4436/4463/ 6313/6287/8173/8192	12645

Tabla 46 - Tiempos prueba 02

Grafica

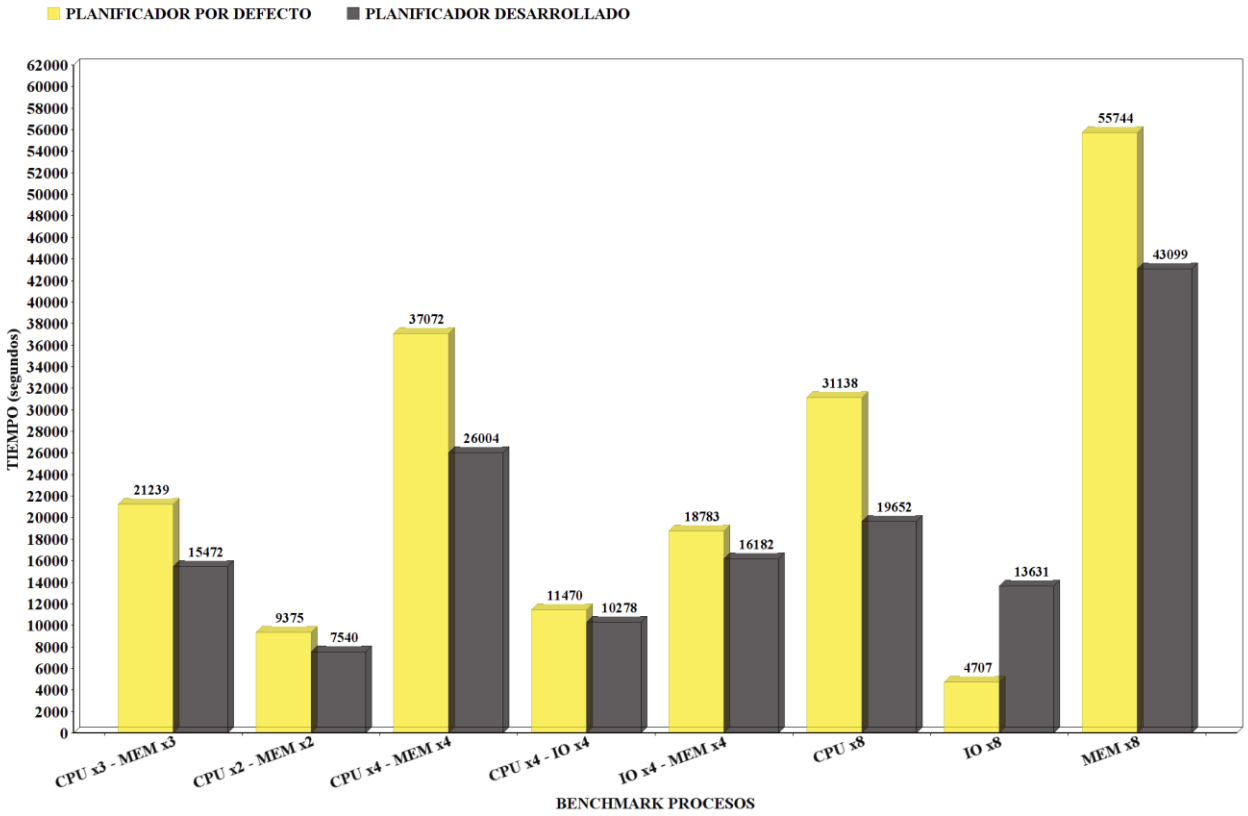


Figura 19 - Gráfica prueba 02

Ganancia/Perdida total

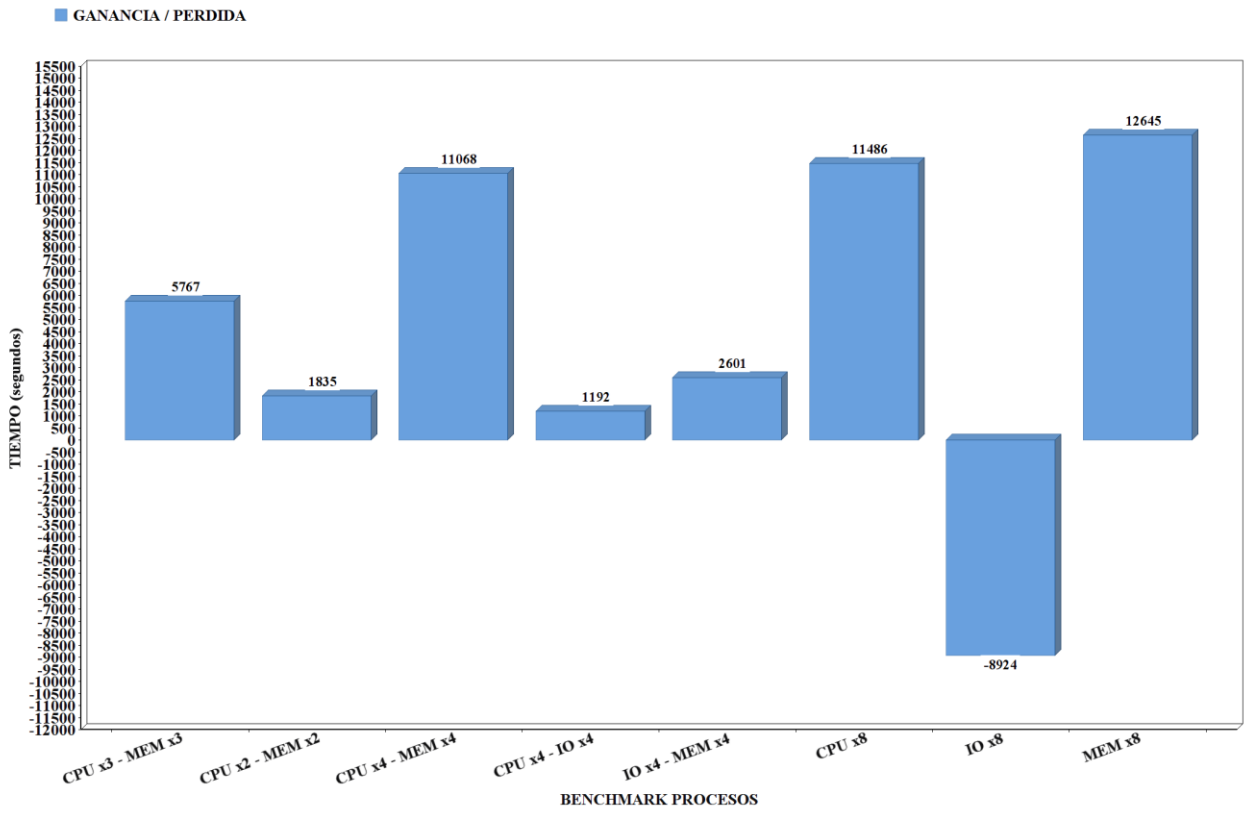


Figura 20 - Gráfica ganancia/pérdida prueba 02

Como podemos observar en las figuras 19 y 20, para este escenario de pruebas, los tiempos han mejorado de una manera considerable, hasta el punto de que en la mayoría de los casos se obtiene una ganancia de tiempo bastante grande.

Podemos destacar que hay un solo caso en el que se genera una pérdida, en la que interfieren solo procesos de IO. Vemos que estos procesos apenas empeoran su rendimiento al ejecutarse al mismo tiempo, lo cual nos produjo la idea de mejorar un aspecto del planificador y generar un nuevo escenario de pruebas, como es el escenario de pruebas 5 en la sección 6.2.3.5.

Dado que para la ejecución de los tipos de *benchmarks* de dos en dos la ganancia es clara, vamos a proponer el tercer escenario, que consiste en combinar los distintos tipos de procesos y ver el comportamiento con y sin el planificador desarrollado.

6.2.3.3 Escenario Prueba - 03

Procesos iniciales y resultados obtenidos

En la tabla 47 y 48, sucede lo mismo que en la tabla anterior, los tiempos están representados en segundos, y están separados por '/' los del mismo tipo, y por un '+' entre cada tipo de benchmark. El primer ejemplo representa un tiempo de 3405, 3408, 3409 y 3408 para los procesos de CPU, un tiempo de 4429, 2283 y 4599 para los procesos de memoria y por último un tiempo de 792, 786 y 789 para los procesos de I/O. Por otra parte, estos resultados se han partido en 2 tablas para facilitar la lectura de los datos.

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/ Perdida (s)
4 procesos CPU + 3 proceso MEMORIA + 3 procesos IO	3405/3408/3409/3408+ 4429/4483/4599+792/786/789	554/975/1438/1909+ 2907/3856/4761+5155/5707/6181	-3935
2 procesos CPU + 5 procesos MEMORIA + 3 procesos IO	3406/3409+4945/5079/ 5056/4933/4988+541/540/539	537/974+1919/2793/ 3700/4530/5438+5964/6443/6905	-5767
2 procesos CPU + 2 procesos MEMORIA + 6 procesos IO	1967/1968+2850/2849+1027/ 1025/1024/1023/1023/1022	524/991+1879/2767 +3159/3636/4113/4625/5122/5600	-16638
5 procesos CPU + 5 procesos MEMORIA + 2 procesos IO	4859/4871/4875/4877/4875+ 6903/6864/6973/6933/6659+928/923	553/1028/1505/1979/2443 +3398/4232/5066/5993/6888+7390/7872	12193
2 procesos CPU + 2 procesos MEMORIA + 8 procesos IO	1995/2002+2897/2888+1013/1012/ 1010/1009/1004/1007/1005/999	550/1024+1996/2896+3391 /3867/4343/4831/5299/5782/6275/6750	-29163
3 procesos CPU + 8 procesos MEMORIA + 1 procesos IO	5365/5365/5362+8451/8356/ 8132/8421/8364/8179/8144 /8174+536	472/940/1432+2413/3366 4254/5182/6096/6948/7768/8596+9118	26264

Tabla 47 - Tiempos prueba 03 - parte 1

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/ Perdida (s)
4 procesos CPU + 8 procesos MEMORIA + 4 procesos IO	5800/5826/5837/5840+9020/9103/ 8882/9102/9033/8941/8731/8873 +530/528/525/523	552/1029/1506/1983+2862/3688/4585 /5493/6461/7315/8246/9173+9697/ 10198/10693/11184	2429
10 procesos CPU + 3 procesos MEMORIA + 3 procesos IO	6301/6310/6314/6315/ 6318/6319/6315/ 6312/6306/6302+7621/7435/ 7590+1219/1218	553/1038/1509/1954/2445/2923/ 3399/3884/4367/4859+5855/6747/ 7659+8164/8673/9176	14990
4 procesos CPU + 4 procesos MEMORIA + 8 procesos IO	3904/3923/3933/3933+ 5390/5458/5500/5512+ 1724/1725/1727/1727/1731/ 1734/1736/1738	544/1009/1490/1981+2818/3612/ 4540/5459+5970/6486/6974/7473/ 7962/8453/8942/9429	-31747
7 procesos CPU + 7 procesos MEMORIA + 2 procesos IO	6783/6792/6797/6799/6804/ 6802/6797+9876/9770/ 9572/9430/9801/ 9867/9881+515/517	557/955/1437/1918/2382/2850/3340+ 4262/5189/6079/6943/7858/8716/ 9541+10102/10593	34081

Tabla 48 - Tiempos prueba 03 - parte 2

Grafica

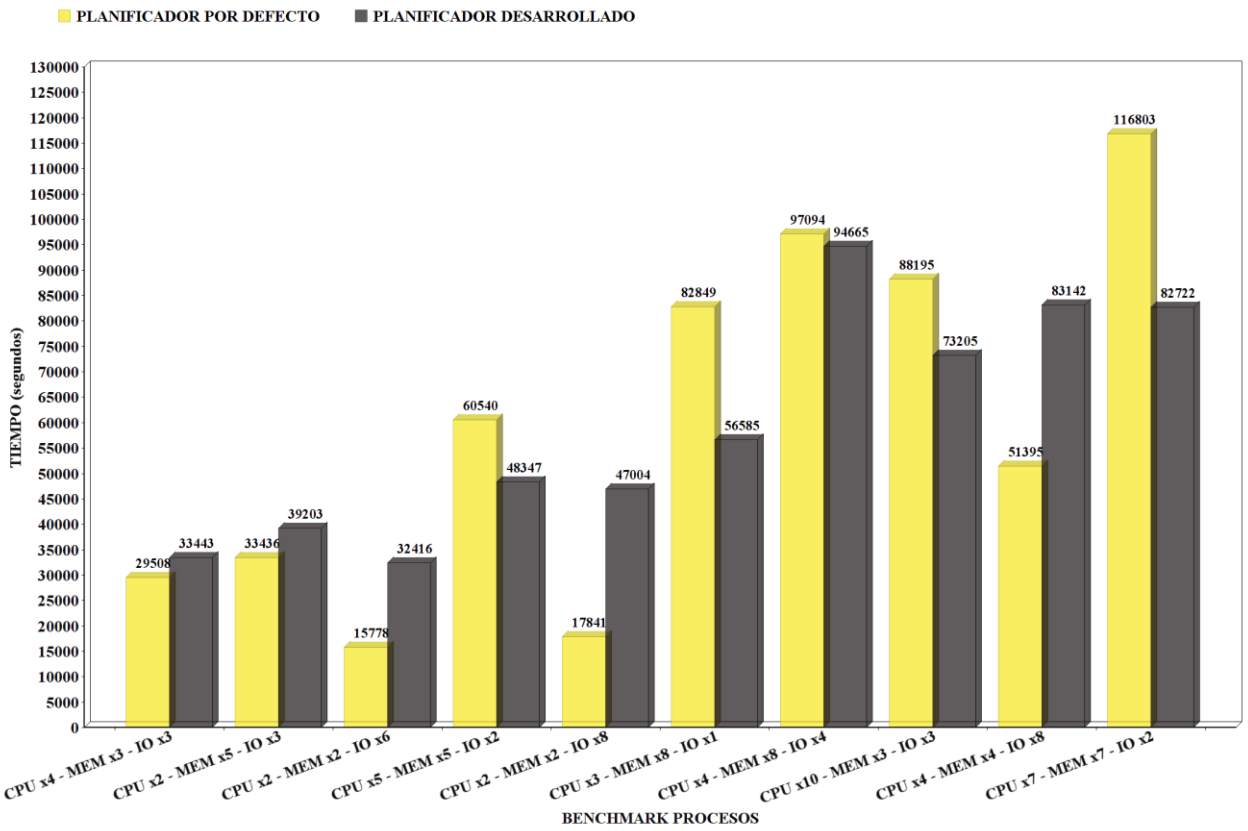


Figura 21 - Gráfica prueba 03

Ganancia/Perdida total

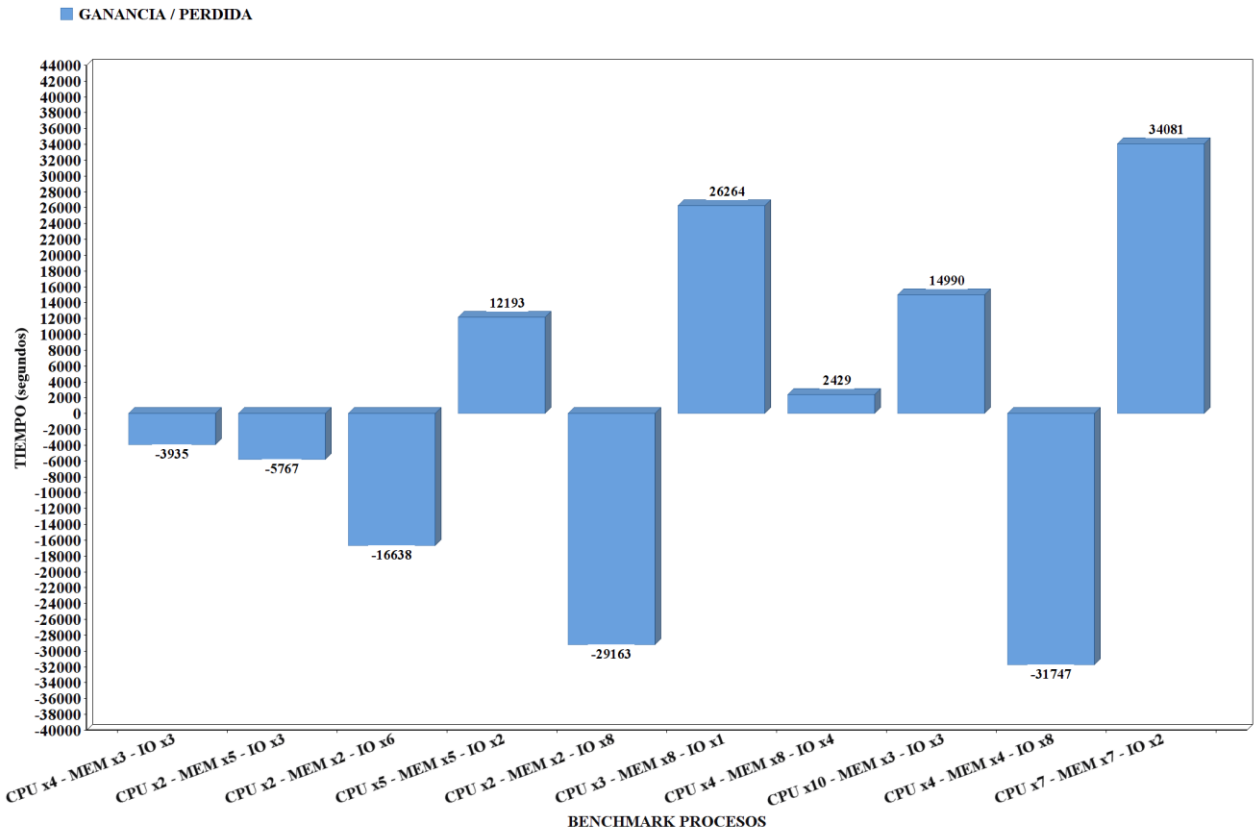


Figura 22 - Gráfica ganancia/pérdida prueba 03

Como observamos en las figuras 21 y 22, la ganancia o la pérdida es muy variada, ya que se llegan a casos en los que la ganancia es muy grande, y casos donde la pérdida es muy grande. Si observamos detenidamente los casos más notorios, podemos observar dos en concreto.

El primer caso es el de la mayor ganancia, cuya batería de pruebas se compone de 7 procesos que ejecutan un *benchmark* de memoria, 7 procesos de CPU y 2 de I/O. Podemos observar en el desglose de tiempos que se produce una gran ganancia al separar la ejecución de los procesos de memoria y los procesos de CPU, ya que cuando se ejecutan tantos procesos de este tipo juntos, el consumo de recursos es tan grande, que el rendimiento es peor que ejecutarlos por separado.

Por otra parte, observamos el peor caso, que es en el que se ejecuta 4 procesos de CPU, 4 de memoria y 8 de I/O. Como ya comentamos en el escenario de pruebas anterior, los procesos de I/O no empeoran el rendimiento al ejecutarlos juntos, ya que la sobrecarga que generan es muy pequeña, lo cual conlleva que cuando hay muchos procesos de este tipo ejecutándose, se pierde tiempo utilizando el planificador diseñado.

Esto nos ha llevado a generar el siguiente escenario de pruebas, en el cual se ejecutan los mismos procesos que en este escenario, pero esta vez se ejecutaran 2 procesos al mismo tiempo, generando el solape que tan buen rendimiento nos dio en el escenario de pruebas 2 en la sección 6.2.3.2.

6.2.3.4 Escenario Prueba - 04

Procesos iniciales y resultados obtenidos

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/ Perdida (s)
4 procesos CPU + 3 proceso MEMORIA + 3 procesos IO	3405/3408/3409/3408+ 4429/4483/4599+792/786/789	1017/1016/3634/1962+ 2631/3646/4463+4150/4672/506	1811
2 procesos CPU + 5 procesos MEMORIA + 3 procesos IO	3406/3409+4945/5079/5056 /4933/4988+541/540/539	1007/1984+1749/2835/4014 /5272/5489+2522/3028/3535	2001
2 procesos CPU + 2 procesos MEMORIA + 6 procesos IO	1967/1968+2850/2849+1027/ 1025/1024/1023/1023/1022	540/1501+2197/2682+520 /2733/3649/3699/4580/4613	-10936
5 procesos CPU + 5 procesos MEMORIA + 2 procesos IO	4859/4871/4875/4877/4875+ 6903/6864/6973/6933/6659+928/923	1505/1506/2954/3901/4877 +2887/3693/5200/6733/6926+5371/7285	7702
2 procesos CPU + 2 procesos MEMORIA + 8 procesos IO	1995/2002+2897/2888+ 1013/1012/1010/1009/1004/1007/1005/999	558/1253+2788/2925+856/845 /3131/3556/4484/4665/5432/5533	-18185

Tabla 49 - Tiempos prueba 04 - parte 1

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/ Perdida (s)
3 procesos CPU + 8 procesos MEMORIA + 1 procesos IO	5365/5365/5362+8451/8356/8132 /8421/8364/8179/8144/8174+536	1025/4060/5050+1461/3086/ 3161/4844/6585/6979/8281/8557+1486	28274
4 procesos CPU + 8 procesos MEMORIA + 4 procesos IO	5800/5826/5837/5840+9020/9103/8882/ 9102/9033/8941/8731/8873+530/528/525/523	1039/2329/3859/4845+1475/ 2878/4594/5545/6823/7712/8583/9051+ 1501/2810/5320/5854	22876
10 procesos CPU + 3 procesos MEMORIA + 3 procesos IO	6301/6310/6314/6315/6318/6319/6315 /6312/6306/6302+7621/7435/7590+1219/1218	1038/1972/1036/5840/6217/6775 /7181/7455/3341/4809+2364/3692/5193+ 2420/7653/3835	17374
4 procesos CPU + 4 procesos MEMORIA + 8 procesos IO	3904/3923/3933/3933+5390/5458/5500/5512 +1724/1725/1727/1727/1731/1734/1736/1738	1260/2632/3580/4507+2471/3061 /4857/5595+540/5440/6284/6446/7221/ 7387/8149/8246	-26281
7 procesos CPU + 7 procesos MEMORIA + 2 procesos IO	6783/6792/6797/6799/6804/6802/6797+ 9876/9770/9572/9430/9801/9867/9881+515/517	1037/1036/2929/3903/4885/5859 /6834+1920/3640/5432/7166/8517/ 8784/9533+1355/1783	42190

Tabla 50 - Tiempos prueba 04 - parte 2

Grafica

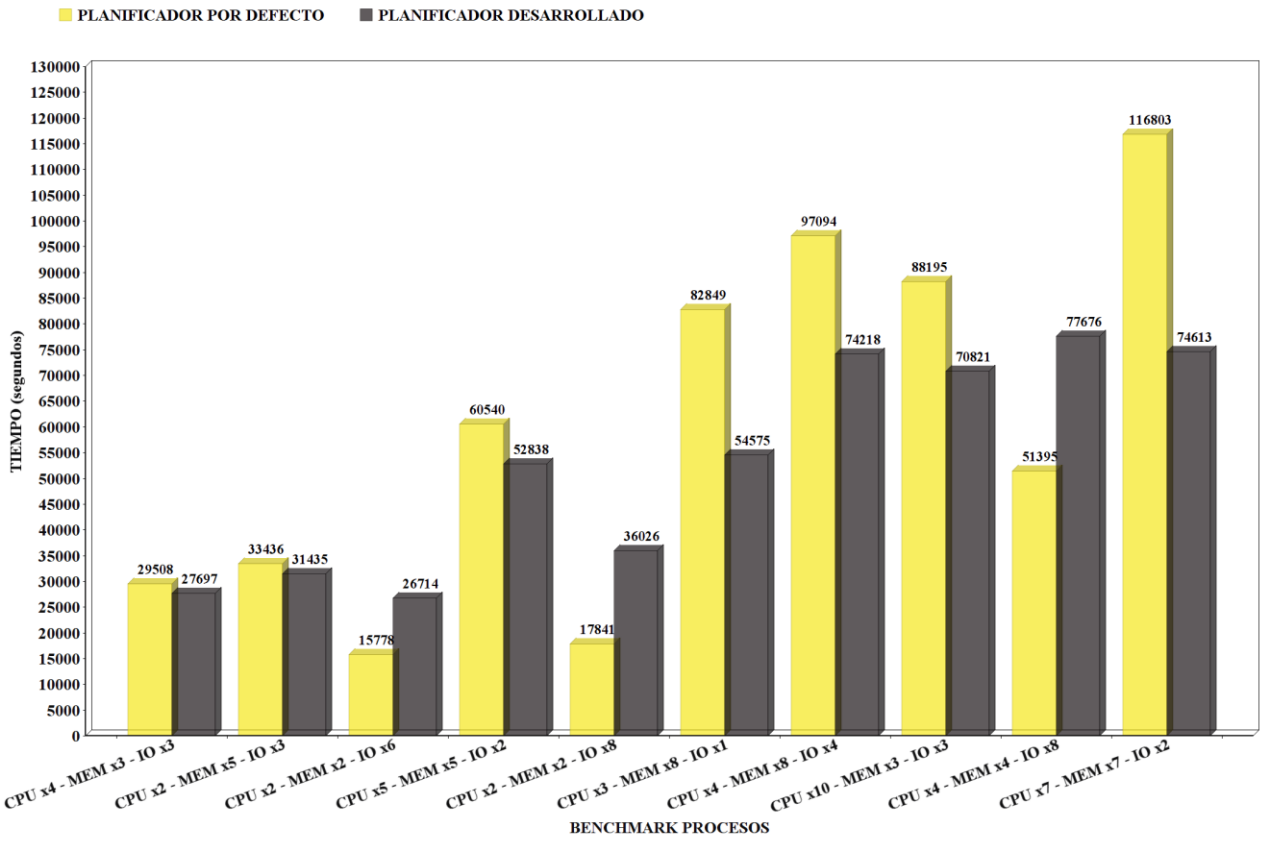


Figura 23 - Gráfica prueba 04

Ganancia/Perdida total

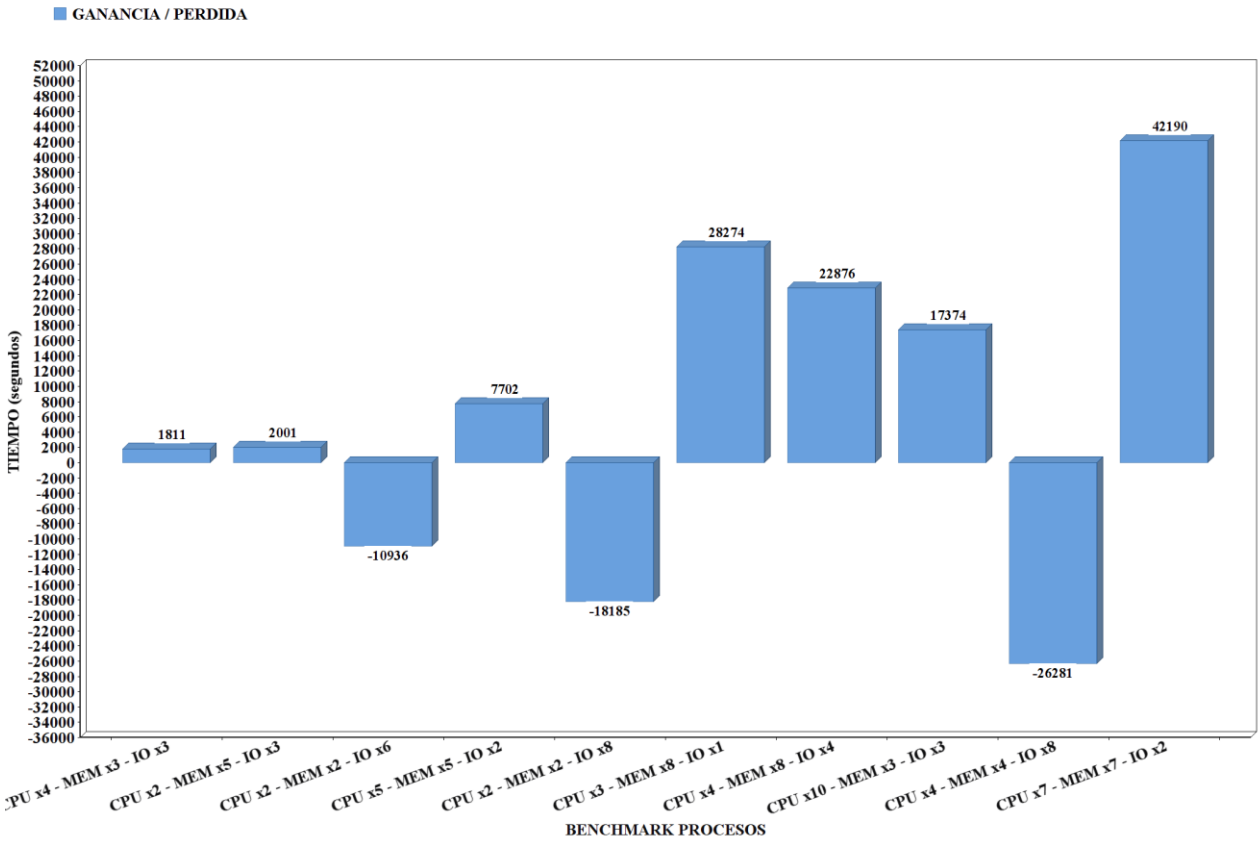


Figura 24 - Gráfica ganancia/pérdida prueba 03

Con esta configuración del planificador, hemos conseguido reducir las pérdidas obtenidas en el escenario anterior, ya que hay casos en los que antes se perdía y ahora ganamos tiempo. Además, en aquellos casos en los que ya había ganancia, se ha aumentado la misma, obteniendo un rendimiento todavía mejor.

Dado que ya hemos mejorado en la mayoría de los escenarios generados el tiempo del planificador que tiene la plataforma por defecto, vamos a tratar de mejorar aquellos casos en los que estamos teniendo pérdidas, ya que si conseguimos esto, el planificador tendrá un rendimiento mejor en todos los escenarios posibles.

Para ellos, hemos generado el escenario de pruebas 5, que consistirá en ejecutar todos los procesos de I/O al mismo tiempo cuando se llegue a un umbral, tratando de evitar la pérdida de rendimiento que se produce al ejecutarlos por separado.

6.2.3.5 Escenario Prueba - 05

Procesos iniciales y resultados obtenidos

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/ Perdida (s)
2 procesos CPU + 2 procesos MEMORIA + 6 procesos IO	1967/1968+2850/2849+1027/1025/ 1024/1023/1023/1022	1039/1051+2618/2699+974 /963/962/960/975/3545	-8
2 procesos CPU + 2 procesos MEMORIA + 8 procesos IO	1995/2002+2897/2888+1013/1012 /1010/1009/1004/1007/1005/999	1012/1008+3184/4104+2362/2347/ 2348/2341/2302/2302/2803/3303	-11576
4 procesos CPU + 4 procesos MEMORIA + 8 procesos IO	3904/3923/3933/3933+5390/5458/5500/5512 + 1724/1725/1727/1727/1731/1734/1736/1738	1030/1028/3615/6460+4468/4932/5973/ 7054+2859/2795/2844/2842/2814/2803/ 4948/5479	-10549

Tabla 51 - Tiempos prueba 05

Grafica

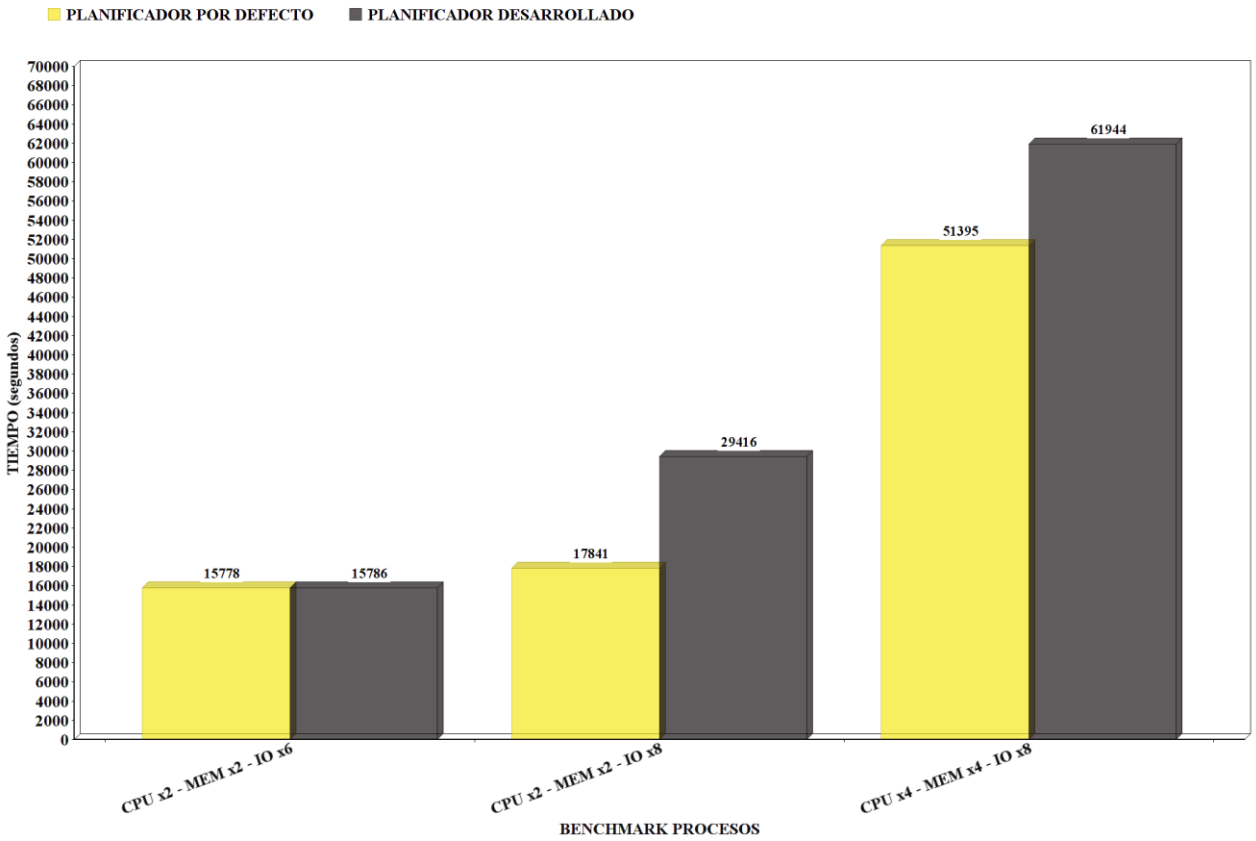


Figura 25 - Gráfica prueba 05

Ganancia/Perdida total

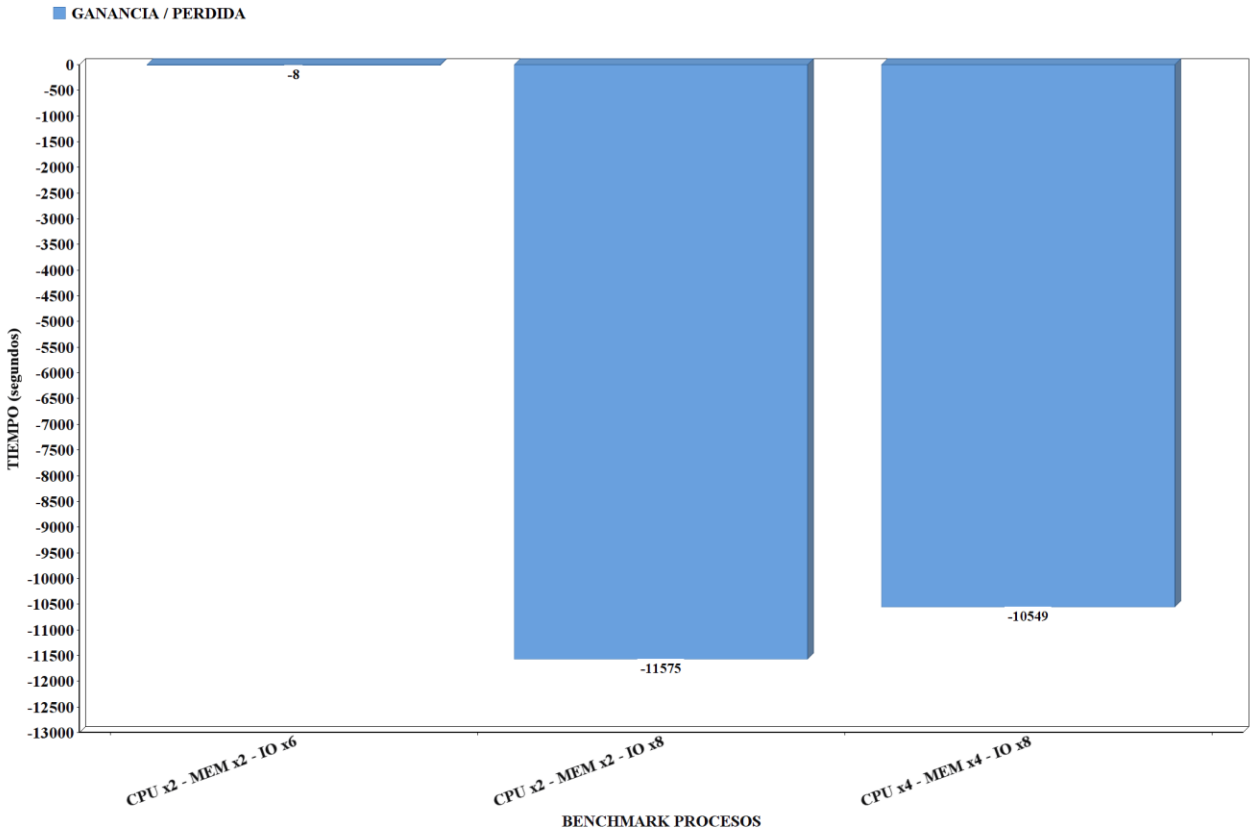


Figura 26 - Gráfica ganancia/pérdida prueba 05

Como podemos ver en las figuras 25 y 26, se ha reducido bastante la pérdida que se producía antes, simplemente ejecutándolos en conjuntos de un determinado número de procesos, que en este caso se ha estipulado en 6 procesos de I/O al mismo tiempo. Esto podría mejorarse aun más si se aumentara el umbral, ya que la pérdida por la ejecución de estos procesos por separado se minimizaría.

Por este motivo, y dado que solamente se genera una pérdida de rendimiento cuando los procesos mayoritarios son de I/O, se ha generado el escenario de pruebas 06, en el cual contamos con que conocemos el número de procesos de I/O y los ejecutamos todos al mismo tiempo.

6.2.3.6 Escenario Prueba - 06

Procesos iniciales y resultados obtenidos

Benchmark de los procesos	Tiempo SIN planificador (s)	Tiempo CON planificador (s)	Ganancia/ Perdida (s)
2 procesos CPU + 2 procesos MEMORIA + 6 procesos IO	1967/1968+2850/2849+1027/1025/ 1024/1023/1023/1022	785/1279+3029/2874+1294/1282/ 1280/1278/1256/1248	173
2 procesos CPU + 2 procesos MEMORIA + 8 procesos IO	1995/2002+2897/2888+1013/1012 /1010/1009/1004/1007/1005/999	548/1049+2400/2755+1456/1447/ 1445/1443/1430/1397/1399/1407	-335
4 procesos CPU + 4 procesos MEMORIA + 8 procesos IO	3904/3923/3933/3933+5390/5458/5500/5512 + 1724/1725/1727/1727/1731/1734/1736/1738	555/1047/1586/3815+3250/3307/ 4601/5166+1415/1417/1411/1413/ 1400/1372/1404/1412	16824

Tabla 52 - Tiempos prueba 06

Grafica

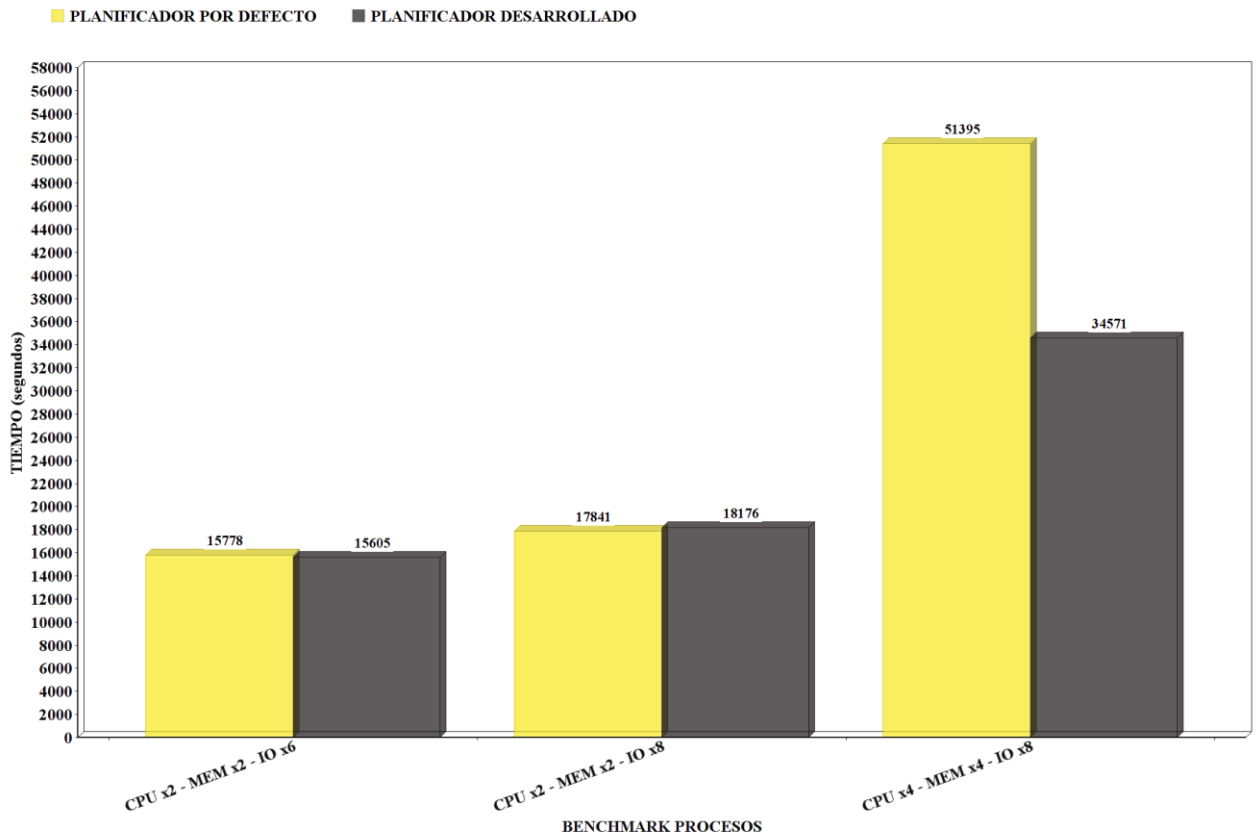


Figura 27 - Gráfica prueba 06

Ganancia/Perdida total

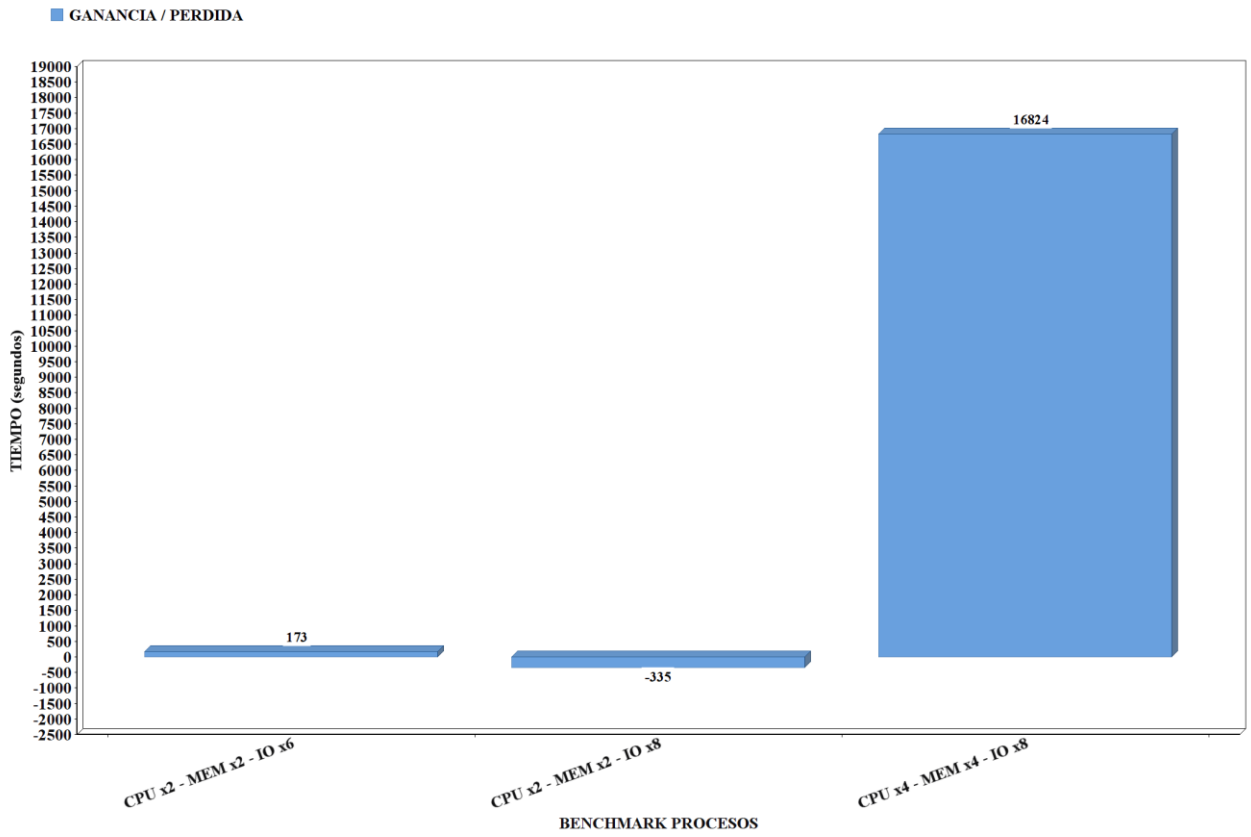


Figura 28 - Gráfica ganancia/pérdida prueba 06

Como observamos en las figuras 27 y 28, en este escenario ideal, estaría solucionado el problema de los procesos de I/O, ya que la ganancia se produciría todos los escenarios producidos menos uno. Esto lógicamente, aunque es un escenario ideal, podría darse el caso de alguna aplicación en la que se conozca a priori todos los tipos de procesos que van a ejecutarse, lo cual nos llevaría a una mejora del rendimiento muy grande.

6.2.3.7 Escenario Prueba - 07

Procesos iniciales y resultados obtenidos

Benchmark	Prioridad	Orden de entrada	Orden de salida	Orden esperado
MEM	1	1	1	Correcto
IO	3	2	7	Correcto
CPU	2	3	3	Correcto
MEM	2	4	5	Correcto
MEM	2	5	6	Correcto
IO	2	6	9	Correcto
CPU	2	7	4	Correcto
IO	3	8	8	Correcto
CPU	3	9	2	Correcto

Tabla 53 - Resultados prueba 07

Para este escenario simplemente se tenía en cuenta que los procesos se ejecutaran en función de su prioridad, y que siguiera la política FIFO si la prioridad de estos coincidía. Este comportamiento es el esperado, de manera que podría introducirse un sistema de prioridades en nuestra aplicación.

Por otra parte, no procede mostrar aquí una gráfica, además de no existir ganancia/pérdida, por lo que esas secciones no se añadirán en este escenario.

6.2.4 Análisis de los resultados

Los resultados obtenidos han sido muy satisfactorios, ya que nuestro planificador ha sido capaz de generar ganancias de tiempo del orden de 40.000 segundos algunos casos, y más de unos 2.000 en casos en los que había un número medio de procesos de cada tipo. Esto nos lleva a analizar las pruebas realizadas en el escenario 5 de nuestro planificador, en el cual observamos que se produce mejora en el 70% de las pruebas, siendo visible que puede producirse una mayor mejora aumentando el umbral de los procesos de I/O a ejecutar.

Lógicamente, si tenemos en cuenta el escenario 6, en el cual supiésemos a priori el número y el tipo de procesos que se van a ejecutar, nos sería aun más favorable el resultado de la implementación de nuestro planificador, ya que solamente se ha producido una pérdida muy pequeña en el 10% de las pruebas, coincidiendo con un gran número de procesos de I/O.

La conclusión que se puede sacar de todas las pruebas realizadas es que los procesos de I/O tienen que ejecutarse en conjuntos, ya que no hay apenas una pérdida existente producida por la sobrecarga de la ejecución de todos estos procesos al mismo tiempo, y a su vez no existe una mejora tangible del rendimiento al ejecutarlos por separado. Además, contra más procesos de CPU y memoria se ejecuten, mayor será la ganancia de nuestra aplicación.

Por último, destacar que el planificador se ha ido mejorando en función de los resultados obtenidos en cada escenario, en los cuales se observaban los resultados y se barajaba que posibilidades había de mejorar el planificador existente, así como distintas baterías de pruebas para probar nuestro planificador.

Las pruebas de prioridad no van enfocadas al rendimiento, pero dan además una posibilidad de uso más para el planificador desarrollado.

Por lo tanto, creo que el modelo propuesto queda validado ya que en los resultados del escenario 5, que son las pruebas con el planificador más avanzado, y con un mayor número de procesos, el porcentaje de mejora supera el 60% de los casos, los cuales presentan distintas configuraciones que se asemejan a un escenario real. Además, si se tiende a aumentar el número de procesos, que es lo más normal en una aplicación real, la mejora será aún mayor.

Capítulo 7: Contexto de desarrollo

7.1 Marco Regulador

Una de las preguntas que pueden surgir a la hora de desarrollar una aplicación software es, ¿Qué es el software ante la ley? A efectos de la presente Ley se entenderá por programa de ordenador toda secuencia de instrucciones o indicaciones destinadas a ser utilizadas, directa o indirectamente, en un sistema informático para obtener un resultado determinado, cualquiera que fuere su forma de expresión y fijación.

Por otra parte, a los mismos efectos, la expresión programas de ordenador comprenderá también su documentación preparatoria. La documentación técnica y los manuales de uso gozarán de la misma protección

Por lo tanto, lo que no se protegería serían ideas, algoritmos, fórmulas matemáticas, principios generales e interfaces de usuario o de aplicación.

7.2 Marco Regulatorio Aplicable al Software Desarrollado

7.2.1 Software de Uso Legal

La utilización del software desarrollado no incumple la legalidad siempre que no se aplique sobre terceros sin su expresa autorización. De otra forma se estaría cometiendo un delito al vulnerar el artículo 197.

La utilización del software adquirido no incumple la legalidad siempre que no se aplique sobre terceros sin su expresa autorización. De otra forma se estaría cometiendo un delito al vulnerar el artículo 197 del Código Penal Español aprobado por la Ley-Orgánica 10/1995, de 23 de noviembre.

7.2.2 Regulación Sobre la Propiedad Intelectual

El sistema desarrollado se atiene a la legislación y normativa relativa a la propiedad intelectual.

- Ámbito nacional.
    - Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual (LPI), regularizando, aclarando y armonizando las disposiciones vigentes en la materia.
    - Ley 2/2011, de 4 de marzo, de Economía Sostenible (Vigente hasta el 02 de Octubre de 2016).

- Ámbito europeo.

    - Directiva 2001/29/CE del Parlamento Europeo y del Consejo, de 22 de mayo de 2001, relativa a la armonización de determinados aspectos de los derechos de autor y derechos afines a los derechos de autor en la Sociedad de la Información.

    - Directiva 2009/24/CE del Parlamento Europeo y del Consejo, de 23 de abril de 2009, sobre la protección jurídica de los programas de ordenador.

- Ámbito internacional.

    - Artículo 27 de la solemne Declaración Universal de Derechos Humanos, que contempla el derecho de toda persona a la protección de los intereses morales y materiales que se derivan de la autoría de las producciones científicas, literarias o artísticas.

7.2.3 Legislación Aplicable al Sistema de Información

El sistema desarrollado se atiene a la legislación y normativa relativa a sistemas de información.

- Ámbito nacional.

    - Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de carácter personal (LOPD).

    - El Real Decreto 1720/2007, de 21 de diciembre, por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999, de 13 de diciembre, de protección de datos de carácter personal.

7.2.4 Normativas y Estándares

Normativas y estándares seguidos durante la construcción del sistema.

- Ámbito internacional.

    - IEEE STANDARD 830-1993 - IEEE Recommended Practice for Software Requirements Specifications.

    - IEEE STANDARD 1016-1987 - IEEE Recommended Practice for Software Design Descriptions.

    - ISO/IEC 12207:2008 establishes a common framework for software life cycle processes, with well-defined terminology, that can be referenced by the software industry.

7.3 Entorno socio-económico

Actualmente vivimos en una sociedad en la que el desarrollo de software es algo usual, ya que cada día se crean nuevas aplicaciones para el mercado, para facilitar tareas cotidianas, para satisfacer necesidades en empresas de distintos sectores o simplemente por entretenimiento.

Si nos fijamos la diferencia que existe actualmente, podemos ser conscientes de que hace 60-70 años, no existían los ordenadores en la vida cotidiana, ya que como mucho, estos existían en grandes corporaciones. La mayoría de la gente apenas sabía lo que era un ordenador, y mucho menos como podía manejarse.

Este cambio ha hecho que muchas facetas de nuestra vida sean más cómodas, ya que hoy en día tenemos presente todo tipo de software en todo tipo de máquinas, ya sean electrodomésticos, video consolas, teléfonos móviles o cosas tan simples como el juguete de un niño.

Es por este motivo, por el cual, cada vez se está invirtiendo más y más dinero en el desarrollo de software, ya que hoy en día, es casi imposible encontrar un negocio que no posea nada electrónico. Hoy en día, las empresas más importantes del mundo, emplean millones de euros en I+D con el objetivo de crear nuevas aplicaciones, mejorar su producto, automatizar procesos…

Por ejemplo, el ranking de empresas en 2016 que más gasto hicieron en I+D fueron las siguientes:

2016: Top 20 R&D Spenders

2016 Rank		2015 Rank	Company	Geography	Industry	R&D Spend ($Bn)*
1	▶	1	Volkswagen	Germany	Automotive	13.2
2	▶	2	Samsung	South Korea	Computing and electronics	12.7
3	▲	7	Amazon	United States	Software and Internet	12.5
4	▲	6	Alphabet	United States	Software and internet	12.3
5	▼	3	Intel Co	United States	Computing and electronics	12.1
6	▼	4	Microsoft	United States	Software and internet	12
7	▼	5	Roche	Switzerland	Healthcare	10
8	▲	9	Novartis	Switzerland	Healthcare	9.5
9	▲	10	Johnson & Johnson	United States	Healthcare	9
10	▼	8	Toyota	Japan	Automotive	8.8
11	▲	18	Apple	United States	Computing and electronics	8.1
12	▼	11	Pfizer	United States	Healthcare	7.7
13	▶	13	General Motors	United States	Automotive	7.5
14	▶	14	Merck	United States	Healthcare	6.7
15	▶	15	Ford	United States	Automotive	6.7

Figura 29 - Ranking empresas 2016

Como podemos observar, el gasto ha sido de un total de 148.8 billones de euros. Una cantidad mucho mayor que en 2005, que fue la siguiente:

Estamos hablando de 53.9 billones de euros más de gasto en un transcurso de 10 años en tecnología. Esto nos hace pensar que en la actualidad, vivimos en una sociedad que cada vez se enfoca más en la tecnología, por lo que hay un aumento considerable del desarrollo de software, ya que todas las tecnologías tienen por detrás un software encargado de controlar la funcionalidad de esa tecnología.

Por esta razón, cada vez es más evidente que este es un sector en auge, con mucha variedad dentro del propio sector, y una gran proyección en años futuros. De hecho, esto se puede demostrar en la sección 1.1, donde se habla del volumen de ventas de la Raspberry Pi, lo que hace que esta vaya creando versiones mejores, con un hardware más potente, y añadiendo cada vez mas puertos a la plataforma. Esto hace que cada vez tenga más usos en proyectos, con más compatibilidades con otras tecnologías.

El único hándicap de que el software se esté extendiendo a una velocidad tan grande, es que hoy en día te obliga a estar actualizado e informado sobre todo lo que rodea a este sector, ya que prácticamente, la sociedad te obliga a avanzar con él, y si trabajas en este sector, con más motivo, ya que se emplea mucho dinero y el hecho de no estar actualizado te puede hacer perder cantidades enormes de dinero.

2005: Top 20 R&D spenders

2005 Rank		2004 Rank	Company	Geography	Industry	R&D Spend ($Bn)*
1	-	-	Sanofi-Aventis	France	Healthcare	9.3
2	-	-	Microsoft	United States	Software and internet	7.8
3	-	-	Pfizer	United States	Healthcare	7.7
4	-	-	Ford	United States	Automotive	7.4
5	-	-	Daimler	Germany	Automotive	7.0
6	-	-	Toyota	Japan	Automotive	7.0
7	-	-	GM	United States	Automotive	6.5
8	-	-	Siemens	Germany	Industrials	6.2
9	-	-	Matsushita (Panasonic)	Japan	Computing and electronics	5.7
10	-	-	IBM	United States	Computing and electronics	5.7
11	-	-	Johnson & Johnson	United States	Healthcare	5.2
12	-	-	GlaxoSmithKline	United Kingdom	Healthcare	5.2
13	-	-	Intel	United States	Computing and electronics	4.8
14	-	-	Volkswagen	Germany	Automotive	4.7
15	-	-	Sony	Japan	Computing and electronics	4.7

Figura 30 - Ranking empresas 2005

Por último, habría que destacar que nuestra aplicación sería inicialmente de código libre, ya que es la mejor manera de ponerla en conocimiento, ya que la gente la utilizaría y serviría para mejorar la aplicación teniendo en cuenta las opiniones acerca de la misma. Pero podría darse el caso de que genere un interés por parte de un proyecto cualquiera y en ese momento si se ganaría dinero, ya que se podrían derivar acciones del proyecto o incluso para el soporte de la propia aplicación.

Capítulo 8: Planificación

Para el desarrollo del proyecto se han realizado dos tipos de planificaciones. La primera, mostrada en la tabla 54, es una planificación ideal, ya que supone total disponibilidad para trabajar en el proyecto.

Por otro lado hay una segunda planificación, mostrada en la tabla 55 y 56, que muestra realmente los plazos teniendo en cuenta mi situación real, en la cual estoy trabajando en una consultora junto con la paternidad, lo cual aumenta los plazos de una manera considerable.

8.1 Planificación ideal

Actividad	Semana Inicial	Duración (semanas)	Semana															
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Planificación	**1**	**0,5**																
Recopilación de información	**1**	**0,5**																
Configuración del entorno de trabajo	**2**	**1**																
Fase de desarrollo	**3**	**6**																
Fase de pruebas	**9**	**4**																
Fase de análisis de los resultados	**13**	**1**																
Desarrollo de la memoria	**14**	**2**																
Fase final de evaluación del proyecto	**16**	**1**																

Tabla 54 - Planificación ideal

La duración total del proyecto sería de **16 semanas**, teniendo en cuenta una disponibilidad casi total para el estudio, ya que cuando se empieza el proyecto, se suelen tener alguna asignatura más, además de los exámenes pendientes. Por este motivo, la duración ideal del proyecto sería de 16 semanas en total.

8.1.1 Planificación ideal – Diagrama de Gantt

Etapas proyecto	Fecha de inicio	Fecha fin	Duración (semanas)
Monitorización del rendimiento en una Raspberry Pi	**15/05/17**	**15/09/17**	**16**
Planificación	**15/05/17**	**18/05/17**	**0.5**
Recopilación de información	**19/05/17**	**22/05/17**	**0.5**
Hardware Raspberry Pi	19/05/17	20/05/17	0.25
Compatibilidad Raspberry Pi	21/05/17	22/05/17	0.25
Configuración del entorno de trabajo	**23/05/17**	**31/05/17**	**1**
Sistema operativo para Raspberry Pi	23/05/17	26/05/17	0.5
Instalación de herramienta en Raspbery Pi	27/05/17	31/05/17	0.5
Fase de desarrollo	**01/06/17**	**15/07/17**	**6**
Lectura contadores Hardware	01/06/17	10/06/17	1.5
Desarrollo cliente-servidor	11/06/17	30/06/17	2.5
Desarrollo planificador	01/07/17	15/07/17	2
Fase de pruebas	**16/07/17**	**15/08/17**	**4**
Generación de pruebas	16/07/17	26/07/17	1.5
Ejecución de pruebas	27/07/17	07/08/17	1.5
Mejora de pruebas	08/08/17	15/08/17	1
Fase de análisis de resultados	**16/08/17**	**23/08/17**	**1**
Análisis resultados	16/08/17	21/08/17	0.75
Validación modelo	22/08/17	23/08/17	0.25
Desarrollo de memoria	**24/08/17**	**07/09/17**	**2**
Fase final de evaluación del proyecto	**08/09/17**	**15/09/17**	**1**

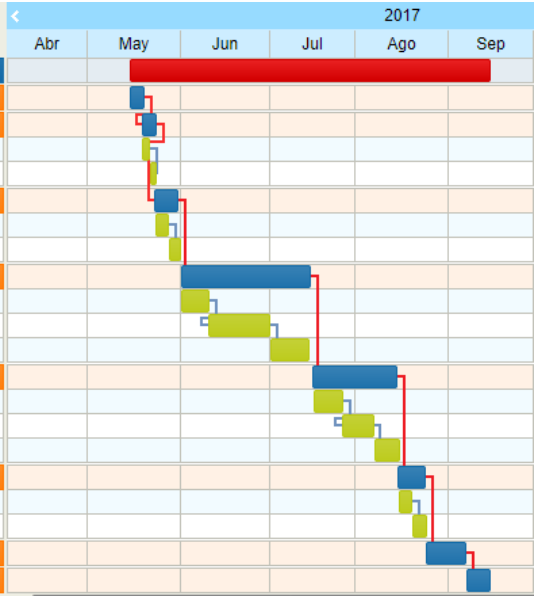


Figura 31 - Diagrama de Gantt planificación ideal

Como podemos observar en la figura 31, existe una dependencia entre cada etapa del proyecto con su etapa anterior. Además, podemos destacar que la fase de desarrollo y la fase de pruebas son las que más tiempo han requerido. Esto se debe a que son las dos partes más importantes del proyecto, puesto que todo va enfocado al desarrollo realizado y las pruebas obtenidas para saber si el modelo desarrollado era válido. Por último, para el resto de fases se ha empleado más o menos el mismo tiempo exceptuando la memoria que si necesitaba más tiempo ya que es el método a través del cual se muestra todo el contenido del proyecto.

8.2 Planificación real

Actividad	Semana Inicial	Duración (semanas)	Semana																					
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Planificación	**1**	**0,5**																						
Recopilación de información	**1**	**2,5**																						
Configuración del entorno de trabajo	**4**	**3**																						
Fase de desarrollo	**7**	**12**																						
Fase de pruebas	**19**	**8**																						

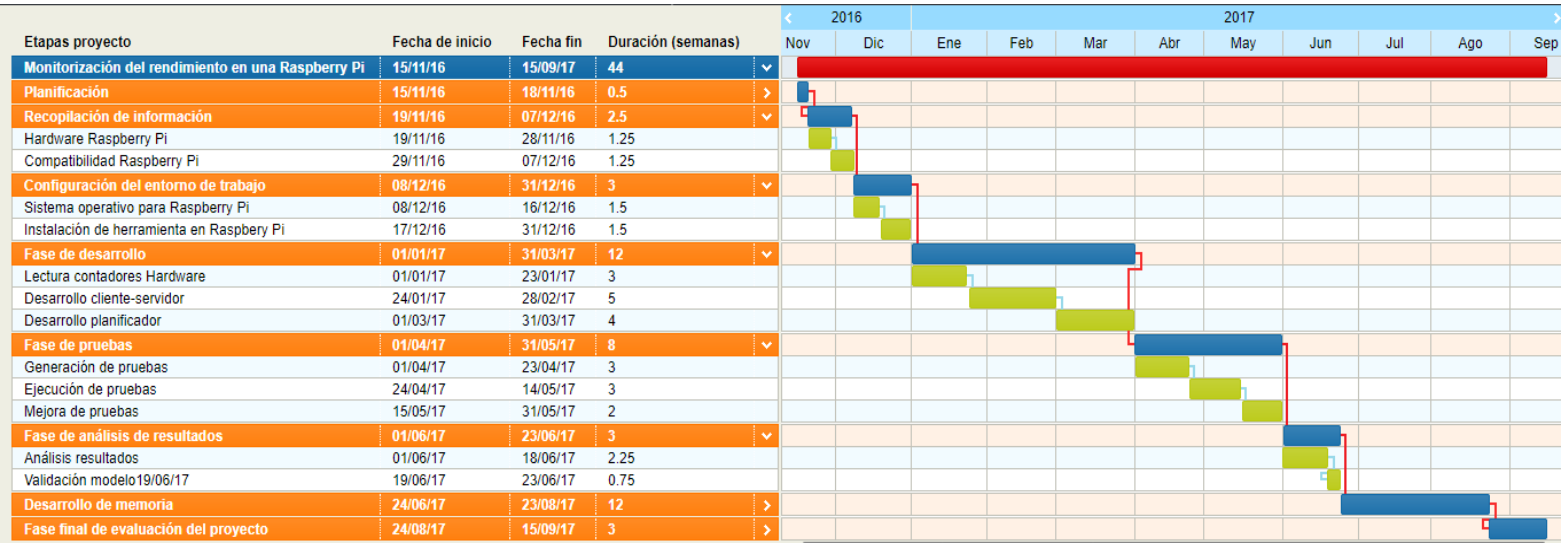
Tabla 55 - Planificación real - parte 1

Actividad	Semana Inicial	Duración (semanas)	Semana																					
			23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
Fase de pruebas	**19**	**8**																						
Fase de análisis de los resultados	**27**	**3**																						
Desarrollo de la memoria	**30**	**12**																						
Fase final de evaluación del proyecto	**42**	**3**																						

Tabla 56 - Planificación real - parte 2

El proyecto tiene una duración de **44 semanas**. Este tiempo supone un aumento considerable respecto a la planificación ideal, pero es la planificación real del proyecto, ya que mi disponibilidad de tiempo es inferior a cuando solamente estudiaba.

8.2.1 Planificación real – Diagrama de Gantt



Etapas proyecto	Fecha de inicio	Fecha fin	Duración (semanas)
Monitorización del rendimiento en una Raspberry Pi	15/11/16	15/09/17	44
Planificación	15/11/16	18/11/16	0.5
Recopilación de información	19/11/16	07/12/16	2.5
Hardware Raspberry Pi	19/11/16	28/11/16	1.25
Compatibilidad Raspberry Pi	29/11/16	07/12/16	1.25
Configuración del entorno de trabajo	08/12/16	31/12/16	3
Sistema operativo para Raspberry Pi	08/12/16	16/12/16	1.5
Instalación de herramienta en Raspbery Pi	17/12/16	31/12/16	1.5
Fase de desarrollo	01/01/17	31/03/17	12
Lectura contadores Hardware	01/01/17	23/01/17	3
Desarrollo cliente-servidor	24/01/17	28/02/17	5
Desarrollo planificador	01/03/17	31/03/17	4
Fase de pruebas	01/04/17	31/05/17	8
Generación de pruebas	01/04/17	23/04/17	3
Ejecución de pruebas	24/04/17	14/05/17	3
Mejora de pruebas	15/05/17	31/05/17	2
Fase de análisis de resultados	01/06/17	23/06/17	3
Análisis resultados	01/06/17	18/06/17	2.25
Validación modelo19/06/17	19/06/17	23/06/17	0.75
Desarrollo de memoria	24/06/17	23/08/17	12
Fase final de evaluación del proyecto	24/08/17	15/09/17	3

Figura 32 - Diagrama de Gantt planificación real

La figura 32 contiene el tiempo empleado en las distintas etapas del proyecto. Si analizamos el tiempo, podemos destacar que cada etapa ha necesitado de mucho tiempo, exceptuando la planificación, la cual era más sencilla cuadrarla por correo o en una reunión en el despacho, puesto que no genera las mismas incertidumbres que pueden generarse en el resto de fases.

Destacar que las etapas que más tiempo han requerido han sido la etapa de desarrollo, pruebas y redacción de la memoria. Estas etapas son las más importantes del proyecto puesto que suponen el grueso del mismo. Es por esto que abarcan casi el 75% del tiempo empleado al proyecto.

Capítulo 9: Presupuesto

En esta sección se va a mostrar un desglose de los costes de este proyecto en el escenario ideal que se había planteado, ya que dada la situación especial en la que se ha desarrollado el proyecto y la duración de este, el presupuesto que obtendríamos sería un caso excepcional y no se ajustaría a un escenario real.

La tabla 57 contiene un desglose de los costes derivados de la contratación de personal. Para ello, hemos tenido en cuenta los siguientes datos:

- 1 Hombre mes: 131,25 horas.
- Máximo anual de dedicación de 12 hombres mes: 1575 horas.
- Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes: 1155 horas.
- Coste hombre mes: Este sería el coste estipulado para el presupuesto de un proyecto en la UC3M dependiendo de la categoría del personal. En estos gastos se incluirían los costes de empresa, como la seguridad social, etc.

Gastos de personal				
Nombre	Categoría	Dedicación (hombre mes)	Coste hombre mes (€)	Coste (€)
David Expósito Singh	Ingeniero Senior	0.5	4.289,54	2.144,77
Álvaro de Lucía Palacios	Ingeniero	3.5	2.694,39	9.430,37
			TOTAL	**11.575,14**

Tabla 57 - Gastos de personal

La tabla 58 muestra los costes empleados en los equipos, así como material utilizado durante el proyecto. Para realizar este cálculo de costes, se sigue la siguiente formula de amortización:

Fórmula de cálculo de la Amortización: A / B * C * D

Donde:

- A: número de meses desde la fecha de facturación en que el equipo es utilizado.
- B: periodo de deprecación.
- C: coste del equipo (sin IVA).
- D: Porcentaje del uso que se le dedica al proyecto (habitualmente 100%).

Por último, habría que hacer un cálculo de los costes directos imputables, como podrían ser viajes, dietas... y que no han sido contemplados anteriormente. En nuestro caso es 0, ya que no hay ese tipo de gastos en nuestro proyecto y al ser software libre no hay gastos de licencia tampoco.

Equipos					
Descripción	Coste (€)	Uso dedicado proyecto (%)	Dedicación (meses)	Periodo de deprecación (meses)	Coste computable (€)
Ordenador portátil	450,00	100	4,00	60	30,00
Raspberry Pi	36,00	70	2,80	60	1,18
Cable HDMI	8,00	70	2,80	60	0,26
Ratón USB	8,00	70	2,80	36	0,44
Teclado USB	9,00	70	2,80	36	0,49
Tarjeta SD	14,00	70	2,80	60	0,46
				TOTAL	**32,83**

Tabla 58 - Costes equipos

El importe total sería de 13.929,56 € (**trece mil novecientos veintinueve euros con cincuenta y seis céntimos de euro).**

Resumen costes	
Concepto	Coste (€)
Personal	11.575,14
Amortización	32,83
Costes Indirectos	2.321,59
TOTAL	**13.929,56**

Tabla 59 - Resumen costes proyecto

La tabla 60 contiene la información general del proyecto, donde incluimos el título del proyecto, así como el autor, fecha, duración, presupuesto y la tasa de costes indirectos que se ha aplicado al sumatorio final del coste.

Información del proyecto	
Título	Monitorización del rendimiento en una Raspberry Pi
Autor	Álvaro de Lucía Palacios
Fecha de comienzo	15 de mayo de 2017
Duración	4 meses
Tasa indirecta	20 %
Presupuesto total	**13.929,56 €**

Tabla 60 - Información del proyecto

Por último, vamos a realizar el cálculo de lo que supondría una venta de nuestro proyecto, teniendo en cuenta los costes computables, así como un margen económico por riesgos y contingencias, el beneficio que se quiere obtener del proyecto y los impuestos.

Coste venta del proyecto			
Concepto	Incremento (%)	Valor Parcial (€)	Coste acumulado (€)
Proyecto	-	13.929,56	13.929,56
Riesgos y contingencias	15,00	2.089,43	16.018,99
Beneficio	30,00	4.805,70	20.824,69
Impuestos	21,00	4.373,19	25.197,88
		Total	**25.197,88**

Tabla 61 - Coste venta proyecto

El precio sería de 25.197,88 € **(veinticinco mil ciento noventa y siete euros con ochenta y ocho céntimos de euro).** Además, si el cliente deseará un mantenimiento de nuestra aplicación, el coste anual sería un 20% sobre el coste acumulado en los beneficios, lo que se traduciría en:

Mantenimiento anual	
Porcentaje sobre el beneficio (%)	Cuota de mantenimiento (€/año)
20%	**4.164,93**

Tabla 62 - Mantenimiento anual

Capítulo 10: Conclusiones y trabajos futuros

En primer lugar creo conveniente enumerar si se han cumplido los objetivos que se plantearon, revisando cada uno de ellos individualmente:

- Creo que se ha realizado un análisis adecuado de los contadores hardware existentes en la plataforma ya que los escogidos han servido para hacer una diferenciación adecuada de los tipos de benchmarks.

- Este objetivo también se cumple ya que los resultados obtenidos son muy satisfactorios, por lo que podemos decir que este objetivo se ha alcanzado.

- Este planificador se ha desarrollado y, aunque no aporta ninguna mejora del rendimiento, nos da una posibilidad más si se decidiera utilizar el planificador en una aplicación real, ya que esto es algo muy usual en las aplicaciones.

- La aplicación se ha modularizado correctamente, separando esta en dos partes claramente diferenciadas, las cuales tienen sus tareas específicas aunque igual de necesarias para el correcto funcionamiento de la aplicación.

El objetivo principal también se habría alcanzado, ya que todos aquellos objetivos que se enfocaran a la mejora del rendimiento del sistema se han cumplido dado que los resultados lo muestran, teniendo ganancia en la mayoría de los test, con ganancias de tiempo muy grandes.

Este proyecto bajo mi punto de vista se ha desarrollado con bastante conocimiento adquirido durante el grado, ya que se compone de un cliente-servidor que se comunica a través de sockets, se ha desarrollado un planificador (con distintas políticas, ya sea FIFO, prioridad…), se explotan los recursos de una plataforma tratando de mejorar el rendimiento utilizando el comando 'perf'… Como es lógico, el haber trabajado ya con todo esto, ha facilitado el desarrollo del mismo y a su vez, ha servido para ampliar y afianzar el conocimiento que ya se tenía en esta materia.

Por otra parte, creo que es un proyecto basado en una tecnología en auge, como es la Raspberry Pi, ya que se han puesto de moda todas estas computadoras de placa reducida, ya que ocupan un espacio muy pequeño, tienen un hardware bastante potente en relación a su coste y ,además, se están mejorando mucho con el tiempo (solo hay que ver las especificaciones del modelo de la Raspberry Pi usada en este proyecto, y las especificaciones de la última Raspberry Pi que ha salido al mercado). Si se consiguiera explotar al máximo la capacidad de estas máquinas, podrían llegar a ser realmente útiles, ya que cada vez están surgiendo distintos usos para estas y se puede acoplar entre ellas para aumentar su potencia de cálculo.

No obstante, al ser una tecnología relativamente nueva, tiene sus limitaciones, las cuales nos han generado algunos problemas durante el transcurso del proyecto, como fue el intento de instalar la librería 'PAPI' para realizar la monitorización de los eventos, en el cual la librería era compatible con la arquitectura, pero no era capaz de obtener los valores de los contadores hardware, por lo que no pudimos utilizarla finalmente, lo cual nos llevo a tener que buscar otras herramientas que

nos proporcionaran la misma funcionalidad que la librería PAPI y que fuera compatible con la arquitectura.

Personalmente, me ha parecido un proyecto entretenido, ya que no era solamente programar una aplicación y probar que el funcionamiento es correcto, sino que había que observar el hardware que se estaba utilizando, valorar como podría mejorarse la aplicación en base a cómo funcionaba por defecto, conocer un poco mas una tecnología relativamente nueva…
Aunque también he de decir, que me ha costado bastante en muchos puntos el desarrollo del mismo, ya que dada mi situación personal, dispongo de poco tiempo, y cuando había contratiempos en el proyecto, me generaba algo de frustración por aumentar más el plazo de entrega de este.

Creo que este proyecto se puede potenciar mucho más en un futuro, teniendo en cuenta las mejoras en el hardware de las Raspberry Pi. Se puede mejorar el planificador para poder darle un uso en aplicaciones que tengan consumo de recursos del sistema de varios tipos, con el fin de mejorar el rendimiento de una manera considerable.

En nuestro caso la plataforma solo tenía un núcleo, pero en las siguientes versiones, se puede aumentar el número de procesos en ejecución, además de tener en cuenta más contadores hardware del sistema para intentar segmentar un poco más la clasificación de los procesos, y acoplarlos de una manera más precisa para aumentar aún más la mejora de rendimiento.

Por último, en trabajos futuros me gustaría probar este planificador en una aplicación real, y a poder ser con varias consolas, ya que de esta manera, podría bombardear al sistema con muchos procesos y ahí se vería realmente la ganancia total y el rendimiento de nuestro sistema. Creo que hoy en día, esta tecnología no está muy explotada, y puede ser algo que se puede aprovechar para conseguir algo novedoso. También se podría realizar esto mismo con *Arduino*, que es una tecnología que he utilizado durante el grado y me ha parecido también muy potente.

Anexo 1: Manual de instalación

- Instalación sistema operativo en tarjeta SD:
    - Descargar sistema operativo NOOBS **[8]**.
    - Descomprimir el fichero comprimido descargado en el punto anterior dentro de la tarjeta SD (tiene que estar formateada).
- Instalación comandos necesarios para el proyecto:
    - Sudo apt-get install linuxtools
    - Instalación *benchmarks:*
        - Descargar 'sysbench' **[5]**.
        - Descomprimir paquete.
        - Ejecutar comandos dentro del paquete:
            - './autogen.sh'
            - './configure'
            - 'make'
            - 'make install'

Anexo 2: Manual de usuario

- Arranque Raspberry Pi:
    - Introducir tarjeta SD en Raspberry Pi.
    - Conectar a la fuente de alimentación la Raspberry Pi.
- Arranque servidor:
    - Ejecutar en la carpeta de la aplicación el comando: ‘./server localhost <puerto>’. Siendo el puerto un puerto libre del sistema.
- Ejecución de los procesos y los clientes:
    - ‘./procesoN’. Siendo N el número de proceso que se va a ejecutar.
    - ‘./clienteN localhost <puerto>’. Siendo N el número de cliente que se ejecuta y el puerto deberá coincidir con el mismo donde se arrancó el servidor.

*Opcional:

- Ejecución de los distintos tipos de benchmarks:
    - Ejecutar comando CPU: ‘./sysbench --test=cpu run’
    - Ejecutar comando memoria: ‘./sysbench --test=memory run’
    - Ejecutar comando E/S: ‘./sysbench --test=fileio --file-total-size=4G --file-num=64 --file-test-mode=rndwr run’

La ejecución de los *benchmarks* por separado es opcional para observar el funcionamiento de los mismos, ya que los procesos contienen la ejecución de un tipo de *benchmark* específico.

Anexo 3: Resumen

Actualmente ha aumentado el uso de consolas como la Raspberry Pi, ya que tienen una calidad-precio bastante buena, además de un gran rendimiento y un tamaño pequeño. Además, esta consola se está mejorando constantemente como podríamos ver comparando las especificaciones del modelo que utilizamos en este proyecto y el último modelo lanzado. Es por esto que cada vez esta aumentado el número de usos para estas consolas, como puede ser en domótica o educación.

Es por esto que se decidió realizar el proyecto con esta tecnología, ya que consideramos que tiene mucho potencial y un futuro interesante. El objetivo que se busco era realizar un planificador de grano grueso capaz de mejorar el rendimiento que tiene la plataforma usando el planificador por defecto.

Para probar el planificador desarrollado se hizo uso de benchmarks para someter al mayor estrés posible al sistema, y generar una sobrecarga suficiente para poder realizar medidas de tiempo y analizar los resultados. Para ello se han utilizado tres familias de benchmarks (CPU, memoria y entrada/salida).

Se han desarrollado varias políticas para el planificador, pero la política con mejores resultados ha sido una política FIFO que ejecuta los benchmarks combinándolos por distinto tipo y ejecutando todos los de E/S en grupos cuando estos llegan a un umbral determinado.

Los resultados obtenidos nos han generado una mejora de tiempos en el 70% de las pruebas. Esta mejora supone una ganancia de tiempo de magnitudes muy grandes. Estamos hablando de que la ganancia existente en estas pruebas puede llegar a los 40 mil segundos, y en los casos que menos se superan los 2 mil segundos. Obviamente estamos hablando de ganancias de tiempo muy grandes si tenemos en cuenta que las ejecuciones son entre 10 y 16 procesos. Si el número de estos aumenta, las ganancias serían aún mayores.

Creemos que este proyecto ha sido satisfactorio ya que se han conseguidos los objetivos propuestos, además de obtener unos resultados bastante buenos. Este proyecto podría continuarse con las nuevas versiones de la Raspberry Pi, ya que mejoran el hardware lo que nos abriría el abanico de posibilidades donde mejorar el sistema.

Por último, sería muy interesante probar el sistema en una aplicación real y poder verificar si existe la posibilidad de mantener una aplicación solamente utilizando Raspberry Pi, ya que el coste del sistema se reduciría de una manera considerable y sería muy fácil integrarlo con otros sistemas que utilicen esta consola. Además, con un hardware mas potente podría hacerse más preciso el filtro para clasificar los procesos y mejorar de nuevo los tiempos obtenidos al mejorar el acople de los procesos del sistema.

Anexo 4: Project Summary

Introduction

Nowadays, the SBC (Single Board Computer) usage is incrementing for personal use and for business use. This increment has many reasons, such as a low cost (compared with PC or laptop), small size and energy-efficient. Also, this technology has an optimum cost-benefit ratio.

This technology (Raspberry Pi and Arduino usually) is being used for home automation, embedded systems, like an interface for another systems or even for education. Furthermore, this technology has a lot of information on the Internet and is improving its hardware and software very fast. For that reason, it has been used in some big projects, achieving really good results. For example, it was used in a facial scanner or for meteorology.

Objectives

Main Objective

The main objective of this Project is developing a coarse-grained scheduler. It will be able to organise the task's execution sequence communicating with a server, improving the performance of the system (which is using the default scheduler). This performance improvement will produce less time spend to finish all the tasks, so this improvement is focused on time (seconds normally). The granularity of the scheduler is coarse-grained because we do not want small slices. We need big intervals because all the tasks have a long duration (we are talking about minutes). Also, this way we reduce the system overload because we reduce the number of communications and synchronisations between processors.

Specific objectives

Regarding the specific objectives, we have the next ones:

- Analyse hardware counters on the platform to know which of them have more influence when the CPU, memory or I/O resources are consumed. This way we would know what counters should we used to sort the different types of tasks.

- Analyse the system and its hardware specifications to define the properly number of tasks on execution, maximising the system throughput without having a big impact on the system. Also, we will evaluate the impact on the system throughput of the combination on execution of the different types of tasks, in order to configure a task sequence priority.

- Configure a priority order on the scheduler, executing those process with a high priority first. If there are two tasks with the same priority, it will use FIFO Schedule (First In, First Out).

- Modularize the application, splitting it into two modules. The first one will be composed of clients and processes, and the second one will be composed of the relation between the server and the clients.

State of art

Monitor tools

There are lots of tools which can be used to monitor the performance of a system, but this tools have more functions apart from this one. Some of these tools will be mentioned below:

- SAR: Allow the user monitoring the system throughput in real time. It finds "bottle necks" too.

- Tcpdump: Allow the user analysing the network packages. With this function the user can catch the packages that are arriving to any port or those which are being exchanged in a TCP communication. It also finds "bottle necks".

- Nagios: Is a free code tool that allows the user monitoring infrastructures. It notices when a server or a database is down, or it can even notice the user when there are space issues in a disk.

- Iostat: This tool is used to get the CPU or I/O information of a disk.

- Mpstat: Recover all the processor information.

- Vmstat: Allows the user getting all the information about the virtual memory.

- PS: Shows all the processes running on the system. It allows filtering by username, hierarchy…

- Free: Recover the physic and virtual memory usage.

- Strace: Is a debugger tool. Shows the system calls, signals… This tool is really powerful if the user does not have the source code.

- Lsof: This tool allow the user monitoring all the open files on the system. This includes network connections, devices and system folders.

As you can see there are a lot of tools, but as our application use hardware counters to sort the different types of tasks, we decided to use another tool. The first one we thought about was PAPI library. This is a monitoring tool very useful to measure the system performance. It can do measures for hardware or software part.

This library is update to be consistent with the new platforms, but in our case, it could not run properly because it was not able to monitor hardware counters in the Raspberry Pi. That issue is produced because PAPI library cannot monitor our Raspbery Pi ARM architecture, so this option was ruled out.

Then we decided to use Perf. It is a Linux monitor tool which is able to find all the Raspberry Pi hardware counters. Also, this tool is consistent with the platform, so this was the option we took. This tool allows the user filtering by PID, time, events… So, after come tests, we realise that this tool was enough to reach our Project goals.

Process scheduling

This is an important part of our Project because the designed scheduler will be in charge of starting/stopping all the pending processes in the server. For that reason, the system overload depends of the scheduler efficiency. In order to choose a right scheduler, we analysed the most popular existing schedulers, as FCFS (First Come, First Served), Round robin or SJB (Shortest Job First).

Once we finished analysing the different schedulers and the requirements of our Project, we decided to design a scheduler which consider the different types of processes and the number of them on execution. This decision was made because our system has only one core and it has a big overload if there are a big amounts of processes running, so we need to find out the balance between the number of running processes and the overload produced by all of them. Also, we added priority to the scheduler. Finally, it was tested to verify it was improving the default one.

Design

Development environment

Our application is written in C language because this language is used for programming systems and we had studied it in the degree, so there was an important knowledge base to speed up the development.

The Operating System we used was Raspbian. It is based on Debian and has an easy installation on the platform. Also, it has an excellent throughput on the platform.

Benchmarks

In order to do an analysis of our platform, we need to reach a high stress on the system. To achieve this, we have used benchmarks of three different types (CPU, memory and In/Out). This benchmarks applications are free code and they are written in C language. In this way, we were able to do a measure of the system throughput.

The different ways this benchmarks have been used to increment the system stress are shown below:

- Combining the different types on execution.
- Executing all in a serial way, all at the same time or executing them in groups.
- Incrementing the number of benchmarks on execution to reach the system limit.

Architecture

The application is divided in two modules. Each one run in a different Raspberry Pi and it is composed of different parts of the system.

The first one runs all the processes and all the clients. The number of processes has no restrictions (except the performance restrictions of the platform) and the number of clients will be the same number as the processes. Each process will run an individual benchmark so it will consume a specific resource of the system.

The process monitoring is a client task. The clients will monitor the process during a short period of time using their PID. To do that, the application use Perf tool to get the information about the different hardware counters. Once it gets all the information, it sends this to the server. That way, the client is not necessary any more, so it can finish the execution. Now the server would be in charge of monitoring the process. Finishing the execution of the client is a design decision made to avoid more overload on the system because if the client was still monitoring the process, the server would need to send signals to the client and then the client send these signals to the process. So this way, we avoid the increment of the communications and also there would be fewer processes on execution and that is less overload for the system.

The processes write their PID on a file, which contains all the processes PIDs. Is from that file where the clients read the PID of the process they will monitor. Once the process do this, it executes their specific benchmark.

The second module is composed of the server and the communication with the clients (this communication use sockets). The server is constantly listening, so if there is a new connection, the server creates a new thread to handle it. That way, the server continue listening and if there is a wrong value in the information that the client has sent, the server doesn't crash. Instead of that, the thread fails and it does not have impact on the application.

This thread will handle all the information and according to it, it will sort the process. Depending on the type, the PID received from the client, will be add to one vector or another one.

Also, the server will create another independent thread in charge of handling all the process execution. So, this thread will do the function of the scheduler, stopping and starting all the pending processes.

In order to avoid concurrency issues (because this application can be configured as a multithread application), the access to the vectors is protected with a mutex. This mutex will block write/read access to the vector because if two different threads access at the same time, they can read the same PID, or they could write the PID in the same position, producing a loose of data.

Implementation

One important part of the creation of the scheduler is choosing the best events to defined which type of process is running. Because of that, we did empirically exhaustive tests with all the events and we came to the conclusion of using the next events:

- L1-dcache-loads: Number of read accesses to level 1 cache memory.
- INSTRUCTIONS: Number of CPU instructions.
- L1-dcache-stores: Number of write accesses to level 1 cache memory.
- CPU-CYCLES: Number of CPU cycles.
- CPU-CLOCK: Processor clock frequency.

With these events, we were able to create the necessary metrics to defined which type of process was running.

Scheduling policy

In order to develop the scheduler, we used some different policies. Some of them have been generated after the experimentation with the previous ones. Once we test one policy, considering the results, we tried to improve it, focusing on those cases were the scheduler was having a bad behaviour.

There are five different policies for the scheduler:

1. FIFO policy. All the incoming processes are executed following a serial sequence, respecting the order they come in. This policy tries to decrease the execution time avoiding the processor overload.
2. FIFO policy executing first a defined type of process. Also, the number of processes in execution is incremented and it always tries to execute together the same type of processes. With this policy we try to get the benefit of overloading the system in a minimal way.

3. FIFO policy combining different types processes. This policy has the benefit that one process consumed more resources of one type and the other consumes another resources of the system. That way, we get a speed up on the execution. Also, this policy overload minimally the processor in order to get better results than the previous policies.

4. FIFO policy including priority. Because of the good results reached with the 3rd policy, we decided to add priority to the processes because in a real application this could be really useful.

5. FIFO policy for all the processes except for I/O ones that will be executed in group when the number of pending I/O process reach a predefined threshold. This way, the execution time is decreased because the I/O processes do not increase the execution time when they are executed together, so we were loosing time executing them in a serial way.

As you can see, there are different policies but all of them are based on FIFO. This is because in a server-client application, if there is no priority, the first incoming client, is the first one who receive the service. However, all the policies have a different point that has been generated taking into account the results obtained executing the previous policy.

Components connection

The communication protocol used between the server and the clients is TCP. In addition, the signals used in the communication between them have been the following ones:

- Connect(): The client establish the connection with the server.
- Accept(): The server keep waiting for new connections.
- Read(): read the value from the socket (It has to be already connected).
- Write(): send a value throw the socket (It has to be already connected).
- Close(): It closes the connection.

The communication between the clients and the processes is not a direct communication, because they communicate throw a file. That is why the signals used in this communication signals to handle files like open(), close(), read() and write().

Finally, the communication between the server and the processes is throw 'kill' signals. We show them below:

- kill (PID, SIGSTOP): This signal stop the execution of the process related to this PID.
- kill (PID, SIGCONT): This signal start the execution of the process related to this PID.

- kill (PID, 0): This signal return value ‘0’ if the process related to this PID exists, This way, we know when a process has finished.

Experimentation

The scheduler has been quite tested. The results obtained have been really important in the evaluation because basing on these results, the scheduler have been improved. Also, we have focused on those cases where there was a big loose of time.

All the tests have been executed 10 times, so the shown result is the average of them. This way, we avoid getting wrong values or unexpected ones.

Every test executed a predefined number of benchmarks of the same type or different one, using the designed scheduler or the default one. Thanks to it, we were able to measure the performance of the system.

The number of different tests executed have been 7, but we will focus on the one we get better results. This test execute group of benchmarks (10, 12 and 16) combining different types of them into this group. This test simulates a high overload on the system incrementing the number of processes.

For this test, the scheduler was configured with the policy 3 explained in the ‘scheduling policy’ section. We get really good results, reducing the execution time in the 70% of the tests. The results are shown in the 32th and 33th figures.

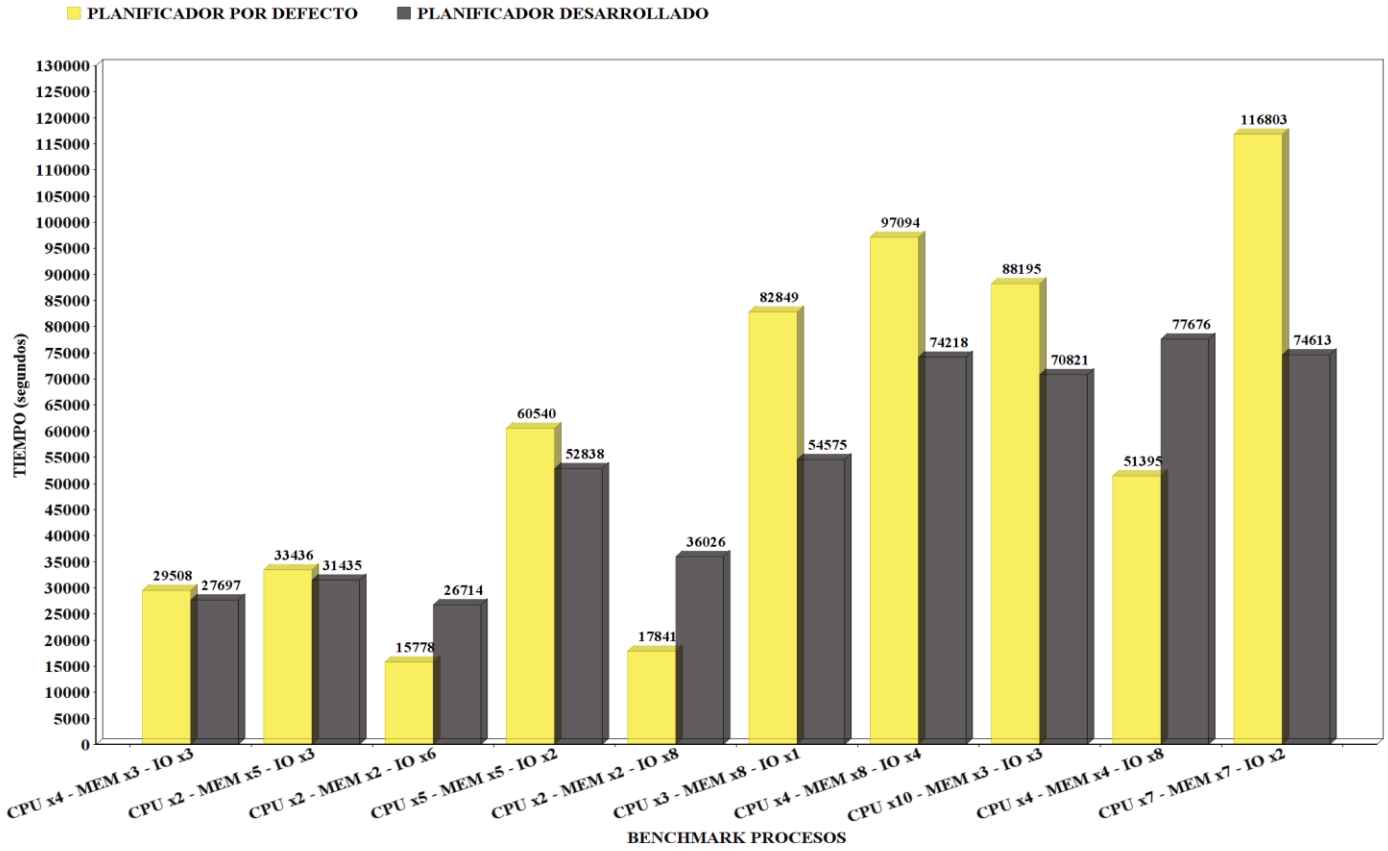


Figura 33 - Results 3rd policy

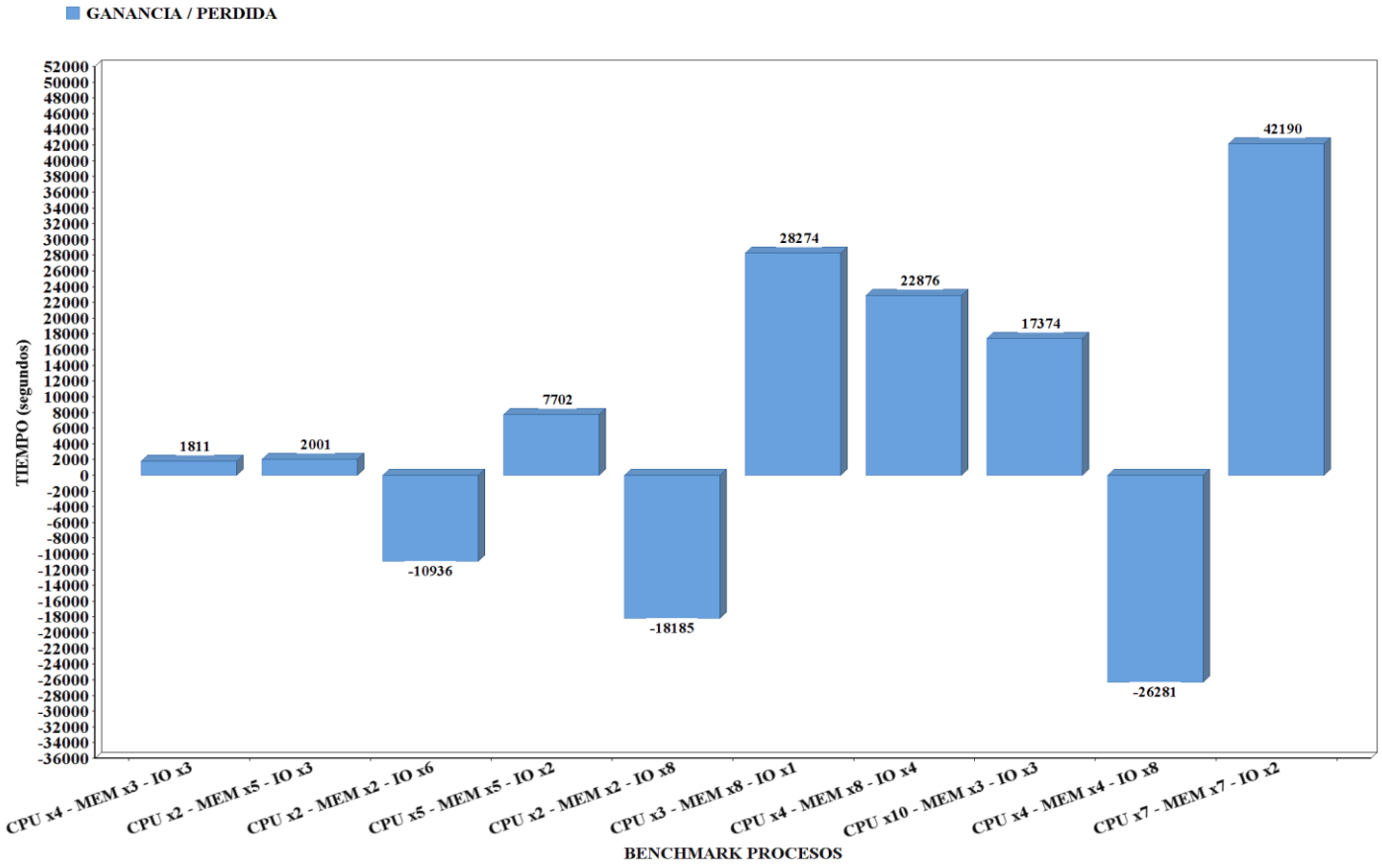


Figura 34 - Profit 3rd policy

However, we analyse the results and we noticed that there was a loss of time when there were more I/O processes on the test group, so that is why we design the 5th policy, testing it again only with the results where we were loosing time. This policy gets really good results, reducing the loss of time to a small period, considering the magnitude we had in the last results. Also, we have to say that all these results were improved again in one test where we supposed that we could know the number of incoming I/O processes. In that case, we would be able to execute them in a big group and the loss of time was reduced again (we had profit in the 90% of the tests). However, we will focus on the test we do not know the number of incoming processes so we will show the results we obtained in the figures 34th and 35th.

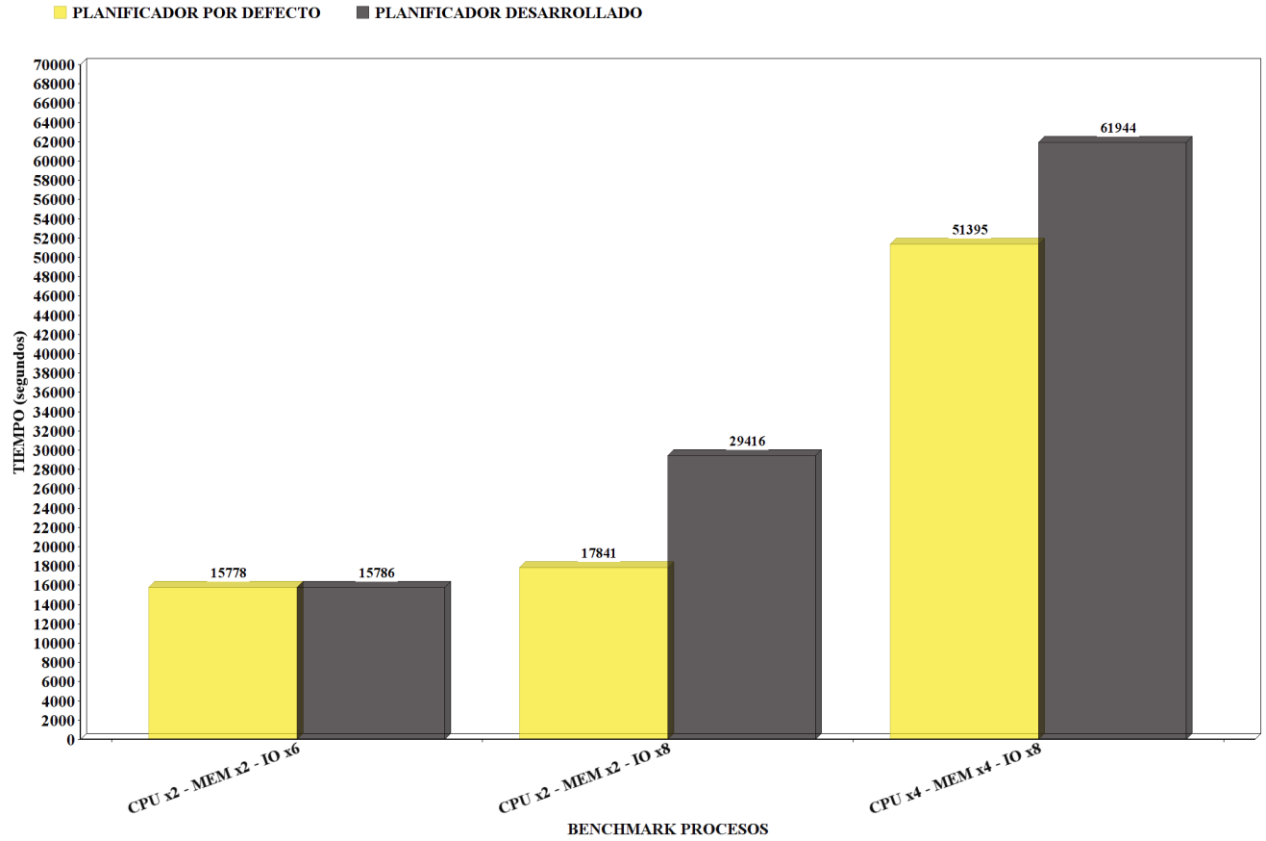


Figura 35 - Results 5th policy

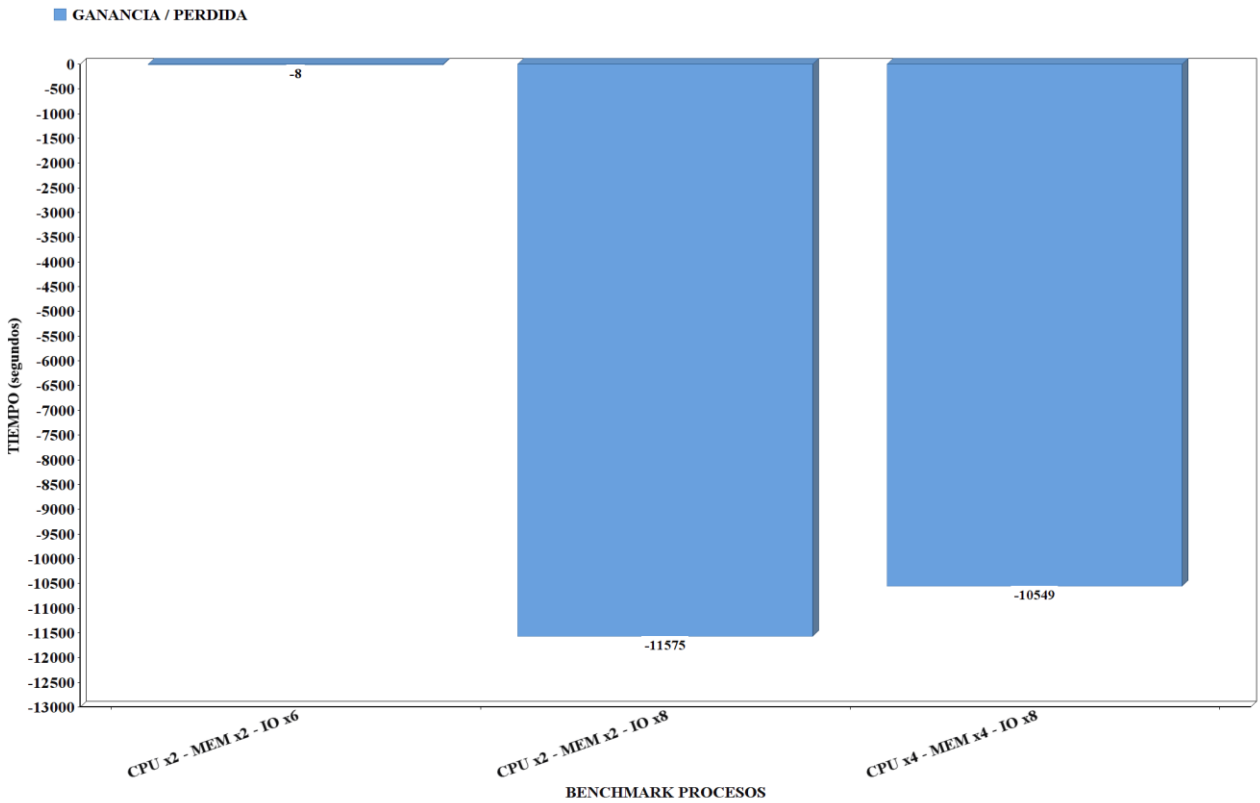


Figura 36 - Profit 5th policy

Model Validation

In order to do a correct validation, we will focus on the objective of this application to consider if the scheduler is properly or not. First of all, it is mandatory taking into account that the objective of this scheduler is improving the total time spent in the execution of the different incoming processes.

Analysing the results, there is a profit of more than 40 thousand seconds in some cases, and more than 2 thousand in those causes where there were an average number of different types of processes. As the 5th test is the one with the last designed scheduler (not considering the one which add priority because is not focus on time), we have profit in the 70% of the tests with the possibility of incrementing this if the number of processes increase, because we have a better behaviour with more processes running than when there are less.

The conclusion of these tests is that all the I/O processes must be executed in groups, because in this way there is no loss of time and there is any fact that these processes have a better behaviour executing them in a serial way.

Underscore that the priority added to the scheduler are not focused on the performance but it gives another use to the designed scheduler and also to the application.

Considering all of this, we validate this model because we were looking for a profit in more than 60% of the tests and we reach this goal and in a real application with probably more processes we would have better results.

Conclusion

This Project has been built with fairly information from the degree. For example the server-client application, the communication with sockets, the scheduler with different policies, the exploit of resources to improve the performance on the system, the use of Perf command… It is normal that all this previous knowledge has supplied the development of this Project.

Also, this is a Project based on a new technology, which is growing too fast (Raspberry Pi), with a powerful hardware and consistent with a lot of new systems (We can compare the specifications of the Raspberry Pi used in this Project and the latest version of it). If we were able to exploit this technology to the maximum it could be a great goal, because they have an easy integration between them and a high estimation power.

However, this is a new technology and it has restrictions that have created some problems when we were choosing for example the monitoring tool. One example was when we install PAPI library and then it was not able to detect all the hardware counters because it was not consistent with the platform ARM version, so we need more time to look for another tool.

Personally I think that is a really interesting Project because it does not consist on programming code and test it. This Project has a theoretical base which is really important when you are making decisions, or you are analysing the behaviour of the system.

Also, I think that everyone likes working with new technologies because you feel that you are not loosing time because then you can use this knowledge in the future.

I do not want to forget that some times I was really upset because the Project was blocked and I do not have enough time to spend on it so it was forwarding slowly. This was one of the reasons why I could not submit this Project on June, because I did not have time to finish the memory. Anyway I am proud of the final result of this Project.

Future works

I think that this Project can be improved in the future because the technology is updated in short periods of time. Considering the new hardware it is using, the scheduler can be improved to increase the different types of processes it can distinguish. Also, it is probably that the performance tools will monitor more hardware counters on it, so the filter can be more accurate.

In our case, the platform only had a core, but this increase in next ones, so the number of threads created to execute the different processes can increase, improving the system performance.

I would like to test this in a real application with some consoles, exploiting all the resources of the system in order to check if it has enough estimation power to keep an application running without another technology. I think that nowadays this technology and Arduino are not exploited enough and it could be nice to give them an opportunity.

Bibliografía y recursos

[1]<<Investigación ARCOS,>>. Recuperado de http://www.arcos.inf.uc3m.es/~desingh/research.html.

[2] <<Librería PAPI,>>. Recuperado de http://icl.cs.utk.edu/papi/.

[3] <<Perf wiki,>>. Recuperado de https://perf.wiki.kernel.org/index.php.

[4] <<Linux performance tools,>>. Recuperado de http://www.thegeekstuff.com/2011/12/linux-performance-monitoring-tools.

[5] <<Repositorio benchmarks GitHub,>>. Recuperado de https://github.com/akopytov/sysbench.

[6] C. Alvarado, D. Tamir and A. Qasem, "Realizing energy-efficient thread affinity configurations with supervised learning," 2015 Sixth International Green and Sustainable Computing Conference (IGSC), 2015. Recuperado de http://ieeexplore.ieee.org/document/7393691/.

[7] J.B. Weissman, L.R. Abburi, D. England, "Integrated Scheduling: The Best of Both Worlds," 2003 Department of Computer Science and Engineering University of Minnesota, 2003. Recuperado de https://www.researchgate.net/publication/222296907_Integrated_scheduling_The_best_of_both_worlds.

[8] <<SSOO Raspberry Pi,>>. Recuperado de https://www.raspberrypi.org/downloads/.

[9]<<Industria del software,>>. Recuperado de https://es.wikipedia.org/wiki/Industria_del_software.

[10]PWC España,<<Informe innovación I+D,>> 2016. Recuperado de https://www.pwc.es/es/consultoria/informe-2016-global-innovation-1000.html.

[11]<<Global Innovation: Top innovators and spenders,>>. Recuperado de https://www.strategyand.pwc.com/global/home/what-we-think/innovation1000/top-innovators-spenders-interactive_only#/tab-2016.

[12] <<Raspberry Pi Model B,>>. Recuperado de http://es.engadget.com/2012/08/11/raspberry-pi-model-b-analizado/.

[13] <<CPU Boards,>>. Recuperado de http://wiki.glidernet.org/cpu-boards.

[14]<<Evolución Raspberry Pi,>>. Recuperado de http://blog.quantika14.com/blog/2016/01/04/la-mochila-del-hacker-iii-raspberry-pi/.

[15]<<ARM 1176JZF S specifications,>>. Recuperado de http://soc.specshero.com/es/qualcomm-msm7200a-vs-arm-1176jzf-s-especificaciones/comparison-2081-846-999.

[16] <<Perf tool information,>>. Recuperado de http://www.brendangregg.com/perf.html.

[17] <<Perf counting,>>. Recuperado de http://www.brendangregg.com/blog/2014-07-03/perf-counting.html.

[18] <<SSOO instalación,>>. Recuperado de https://raspberryparatorpes.net/instalacion/noobs-paso-a-paso-instalar-el-sistema-operativo-en-la-raspberry-pi/.

[19] <<Ventas Raspberry Pi,>>. Recuperado de https://www.xataka.com/makers/la-raspberry-pi-ha-vendido-ya-mas-unidades-que-las-que-logro-vender-el-mitico-commodore-64.

[20]<<Algoritmos planificación,>>. Recuperado de https://www.ecured.cu/Planificaci%C3%B3n_de_procesos_en_un_sistema_operativo.

[21] <<SAR,>>. Recuperado de http://mundogeek.net/traducciones/midiendo-el-rendimiento-del-sistema-con-SAR.htm.

[22] <<Tcpdump,>>. Recuperado de http://www.tcpdump.org/tcpdump_man.html.

[23] <<Nagios,>>. Recuperado de https://www.nagios.org/.

[24] <<Iostat,>>. Recuperado de http://www.admin-magazine.com/HPC/Articles/Monitoring-Storage-with-iostat.

[25] <<Mpstat,>>. Recuperado de http://systemadmin.es/2009/11/como-leer-el-uso-de-cpu-en-el-top-o-el-mpstat.

[26]<<VMstat,>>. Recuperado de (https://docs.oracle.com/cd/E24842_01/html/E23086/spmonitor-22.html.

[27] <<PS,>>. Recuperado de http://www.linfo.org/ps.html.

[28] <<Free,>>. Recuperado de http://www.linfo.org/free.html.

[29] <<Strace,>>. Recuperado de https://strace.io/.

[30] <<Lsof,>>. Recuperado de http://rm-rf.es/linux-el-comando-lsof/.